ascens

www.ascens-ist.eu

ASCENS

Autonomic Service-Component Ensembles

JD3.2: Software Engineering for Self-Aware SCEs

Ensemble Development Life Cycle

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: **LMU**
Author(s): **Nora Koch, Matthias Hölzl, Annabelle Klarl, Philip Mayer (LMU), Tomas Bures (CUNI), Jaquez Combaz (UJF-Verimag), Alberto Lluch Lafuente, Rocco De Nicola, Stefano Sebastio, Francesco Tiezzi, Andrea Vandin (IMT), Fabio Gaducci, Valentina Monreale, Ugo Montanari (UNIPI), Michele Loreti (UDF), Carlo Pinciroli (ULB), Mariachiara Puviani, Franco Zambonelli (UNIMORE), Nikola Šerbedžija (Fraunhofer), Emil Vassev (UL)**

Reporting Period: **3**
Period covered: **October 1, 2012 to September 30, 2013**
Submission date: **November 8, 2013**
Revision: **Final**
Classification: **PU**

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

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

SEVENTH FRAMEWORK PROGRAMME

Executive Summary

To help developers address the challenges posed by the diversity of self-* properties and the engineering of adaptive behaviours, the ASCENS project has defined the ensemble development life cycle (EDLC). In contrast to more classical software development life cycles, in order to guarantee adaptivity, we rely more on the feedback of runtime data to the design phases. We illustrate how the life cycle can be instantiated using specific languages, methods and tools developed within the ASCENS project. The examples include one scenario of each ASCENS case-study field, i.e., cloud computing, e-mobility and swarm robotics, and show how various ASCENS languages and tools, such as SOTA, MESSI, (j)DEECo, SCEL, jRESP, Helena and *Iliad* can be applied in practice.

Although verification and validation issues are part of the EDLC, they are only sketched in this deliverable. The reader is referred for these aspects of the ensembles development life cycle to deliverable JD3.1.

Contents

1 Introduction

The main aim of the ASCENS project is to tackle engineering issues, such as designing, analyzing and control massively distributed and highly dynamic autonomic systems. One of the main challenges for software engineers is then to find reliable methods and tools to build the complex software required by these systems. We propose an engineering approach based on service components and ensembles which implement self-* features.

In this deliverable we present the Ensemble Development Life Cycle (EDLC) that covers the full design and runtime aspects of autonomic systems. It is a conceptual framework that defines a set of phases and their interplay mainly based on feedback loops as shown in Figure 1. The life cycle comprises a "double-wheel" and two "arrows" between the wheels providing three different feedback loops: (1) at design time, (2) at runtime and (3) between the two of them.

Feedback loop at design time (1) enables continuous improvement of models and code due to changing requirements and results of verification or validation. *The control feedback loop at runtime* (2) implements self-adaptation based on awareness about the system and its environment. Finally, *the feedback loop between runtime and design time "wheels"* (3) provides the mechanisms to change architectural models and code on the basis of the runtime behaviour of the continuous evolving system.

Design issues of the EDLC are grouped in requirements, modeling, programming, verification and validation phases. They comprise activities performed offline. EDLC runtime focus on monitoring, awareness and self-adaptation of ensemble-based software systems and are performed online.

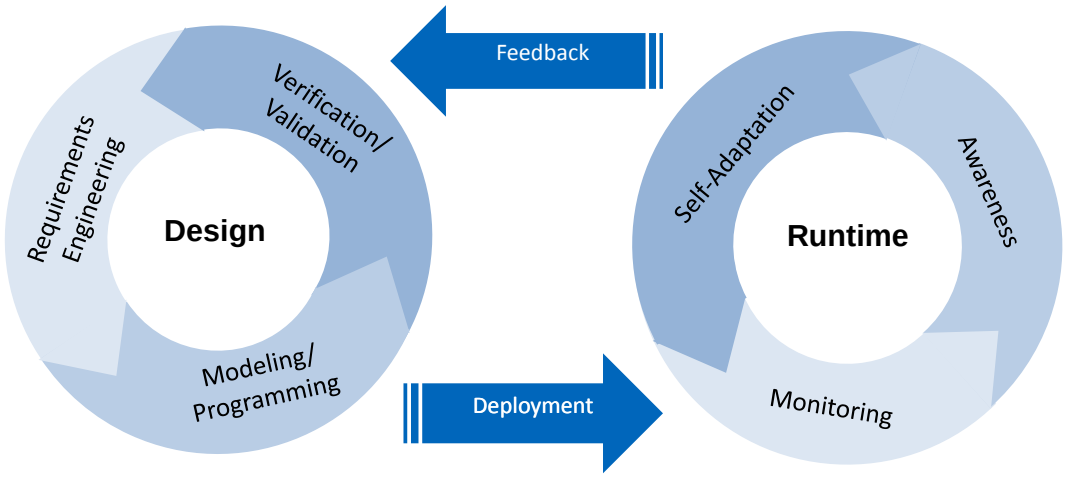


Figure 1: Ensembles Development Life Cycle (EDLC)

We illustrate the EDLC using methods and tools, mostly developed within the ASCENS project. Examples are SOTA for requirements engineering of awareness and adaptive issues, SCEL, Helena and SACPL for modeling different aspects of autonomic systems and at different levels of abstractions, SPL for monitoring, Iliad as awareness-engine, soft-constraints for adaptation, jDEECo and jRESP as runtime frameworks. These methods and tools are specifically designed to capture the self-* features of autonomic systems.

Structure of the deliverable. Section 2 provides an overview of the EDLC. Section 3 presents the example scenarios of the three case studies covering offline, online and transitions between them. Section 4 concludes and sketches our future plans in the context of EDLC.

2 Ensemble Development Life Cycle

The development of ensemble-based systems goes beyond addressing the classical phases of the software development life cycle like requirements elicitation, implementation and deployment. Engineering autonomic systems has to tackle aspects like self-awareness and self-adaptation. Such properties have to be considered from the beginning of the development process, i.e. during elicitation of the requirements. We need to capture how the system should be adapted and how the system or environment should be observed in order to make adaptation possible.

Models are usually built on top of the elicited requirements, mainly in following an iterative process, in which also validation and verification in early phases of the development are highly recommended, in order to mitigate the impact of design errors. In the literature we find several approaches for possible architectures or reference models for adaptive and autonomic systems. A well known approach is the MAPE-K architecture introduced by IBM [Cor05] which comprises a control loop of four phases Monitor, Analyse, Plan, Execute. MAPE-K – in contrast to our approach – focus only on the adaptation process at runtime and does not consider the interplay of design and runtime phases. The second research roadmap for self-adaptive systems [dLea11] also suggests a life cycle based on MAPE-K and proposes the use of a process modeling language to describe the self-adaptation workflow and feedback control loops.

The approach of Inverardi and Mori [IM10] shows foreseen and unforeseen context changes which are represented following a feature analysis perspective. Their life cycle is also based on MAPE-K, focusing therefore on the runtime aspects. A slightly different life cycle is presented in the work of Brun et al. [BMSG$^+$09] which explores feedback loops from the control engineering perspective; feedback loops are first-class entities and comprise the activities collect, analyse, decide and act.

Bruni et al. [BCG$^+$12a] presented a control data based conceptual framework for adaptivity. In contrast to our pragmatic approach supporting the use of methods and tools in the development life cycle, they provide a simple formal model for the framework based on a labelled transition system (LTS). In addition, they provide an analysis of adaptivity in different computational paradigms, such as context-oriented and declarative programming from the control data point of view.

Šerbedžija and Fairclough [SF09] proposes a process of adaptation that is achieved by creating a biocybernetic loop that may operate on several, simultaneous timescales (minutes/hours/weeks/months/years). In terms of architecture, they argued that a sense-analyse-react system requires middleware with closed-loop control consisting of: (1) a tangible layer concerned with sensors and actuators, (2) a reflective layer containing a flexible representation of the user to guide system adaptation, and (3) an application layer representing application scenarios and the context for adaptation and evolution.

Our aim is to focus on these distinguishing characteristics of autonomic systems along the whole development cycle. A relevant issue is then the use of modeling and implementation techniques for adaptive and awareness features. We propose a "double-wheel" life cycle for autonomic systems to sketch the main aspects of the engineering process as shown in Figure 1. The "first wheel" represents the *design* or *offline* phases and the second one represents the *runtime* or *online* phases. Both wheels are connected by the transitions *deployment* and *feedback*.

The offline phases comprise *requirements engineering*, *modeling and programming* and *verification and validation*. We emphasize the relevance of mathematical approaches to validate and verify the properties of the autonomic system and enable the prediction of the behaviour of such complex systems. This closes the cycle providing feedback for checking the requirements identified so far or improving the model or code.

The online phases comprise *monitoring*, *awareness* and *self-adaptation*. They consist of observing the system and the environment, reasoning on such observations and using the results of the analysis for adapting the system and providing feedback for offline activities.

Transitions between online and offline activities can be performed as often as needed throughout the system's evolution, and data acquired during monitoring at runtime are fed back to the design cycle to provide information to be used for system redesign, verification and redeployment.

The process defined by this life cycle can be refined providing details on the involved stakeholders, the actions they perform as well as needed input and the output they produce. A process modeling languages can be used to specify the details. We can use therefore either general workflow-oriented modeling languages such as UML activity diagrams[1] or BPMN[2], or Domain Specific Languages (DSL) such as the OMG standard Software and Systems Process Engineering Metamodel (SPEM)[3] or the Multi-View Process Modeling Language (MV-PML) developed by NASA [BLRV95]. However, describing the refinement and modeling process in detail goes beyond the scope of this overview paper.

Within the ASCENS project several languages, methods and tools have been developed or previously existing ones have been extended to address engineering of ensembles. The development of a particular autonomic system will imply the selection of the most appropriate languages, methods and tools, i.e. an instantiation of the life cycle.

2.1 Designing Self-Aware Systems

The "first wheel" representing the *design* or *offline* phases comprise *requirements engineering*, *modeling and programming* and *verification and validation*. In ASCENS we propose a goal-oriented requirements engineering approach for the identification and modeling of functional and adaptive requirements of autonomous systems. The approach is called SOTA, which stands for "state of the affairs". The self-* properties identified in SOTA are then modeled as cooperating components in the SCEL language providing a seamless transition to implementation in jRESP for example. ASCENS focus also on the mathematical approaches to validate and verify the properties of the autonomic system. Formal methods are used as well to predict the behaviour of such complex systems. This closes the *design* cycle providing feedback for checking the requirements identified so far or improving the model or code.

2.1.1 Requirements Engineering

Traditionally, software engineering divides requirements in two categories: functional requirements (what the system should do) and non-functional requirements (performance, quality of service, etc.). In the areas of adaptive and open-ended systems, both functional and non-functional requirements are better expressed in terms of "goals" [MCY99]. A goal, in most general terms, represent a desirable state of the affairs that an entity, that is a software component or software system, aims to achieve.

In ASCENS we propose SOTA for capturing and specifying the requirements of autonomic systems. SOTA is an extension of existing goal-oriented requirements engineering approaches that integrates elements of dynamical systems modeling to account for the general needs of dynamic self-adaptive systems and components.

SOTA models the entities of a self-adaptive system as n-dimensional space $\mathbf{S}$, with each dimension representing a specific aspect of the current situation of the entity/ensemble and of its operational environment. As an entity executes, its position in S changes either due to its specific actions of or because of the dynamics of environment. Thus, we can generally see this evolution of the system as a movement in $\mathbf{S}$.

[1] UML website: `http://www.uml.org/`

[2] BPMN website: `http://www.omg.org/spec/BPMN/2.0/`

[3] SPEM website: `http://www.omg.org/spec/SPEM/2.0/`

In this context, a goal in SOTA can be expressed in terms of a specific state of the affairs to aim for, that is, a specific point or a specific area in $\mathbf{S}$ which the entity or the system as a whole should try to reach, despite the fact that external contingencies can move the trajectory farther from the goal.

Along this lines, the activity of requirements engineering for self-adaptive systems in SOTA implies: *(i)* identifying the dimensions of the SOTA space, which means modeling the relevant information that a system/entity has to collect to become aware of its location in such space, a necessary condition to recognize whether it is correctly behaving and adapt its actions whenever necessary; *(ii)* identifying the set of goals for each entity and for the system as a whole, which also implies identifying when specific goals gets activated and any possible constraint on the trajectory to be followed while trying to achieve such goals.

The SOTA modeling approach is very useful to understand and model the functional and adaptation requirements, and to check the correctness of such specifications (as described in [AZ12]). However, when a designer considers the actual design of the system, SOTA can help identifying it is important to identify which architectural schemes need to be chosen for the individual components and the ensembles.

To this end, in previous work [CPZ11], we defined a taxonomy of architectural patterns for adaptive components and ensemble of components. At the center of our taxonomy is the idea that self-adaptivity requires the presence of a *feedback loop* or control loop. A feedback loop is the part of the system that allows for monitoring, recognizing the need for adaptation, and putting adaptation actions in place.

However, when it comes to choosing among a variety of possible architectural schemes that can be defied for feedback loops [CPZ11] it becomes clear that the specific characteristics of goals identified in the requirements engineering phase directly guides the choice of specific feedback loop patterns. In particular, the choice of a specific pattern depends on whether (and to which extent) the components of the system have component-specific goals with different characteristics, or whether they share the same ensemble-level goals. That is, the modeling of SOTA goals directly drives the adoption of specific architectural patterns, thus making SOTA a very useful tool for designers.

Concerning our work on knowledge representation, one of the biggest issues was finding the right level of abstraction and data relevance for the knowledge models specified with KnowLang. Past experience demonstrated that our knowledge models carried unnecessary details and often irrelevant data. To meet this challenge, we used the Autonomy Requirements Engineering (ARE) [VH13] approach [4]. that intents to help engineers tackle the integration and promotion of autonomy in software-intensive systems. ARE combines *generic autonomy requirements* (GAR) with *goal-oriented requirements engineering* (GORE).

The ARE approach starts with the creation of a *goals model* that represents system objectives and their interrelationships. For this, we use GORE where ARE goals are generally modeled with intrinsic features such as type, actor, and target, with links to other goals and constraints in the requirements model. The next step is to work on each one of the *system goals* along with the elicited *environmental constraints* to come up with the self-* objectives providing the autonomy requirements for this particular system's behavior. In this phase, we apply our GAR model to a system goal to derive autonomy requirements in the form of goal's supportive and alternative self-* objectives along with the necessary capabilities and quality characteristics.

ARE relies on KnowLang [VHM+12] for the formal specification of the elicited autonomy requirements. Therefore, we use KnowLang to record these requirements as knowledge representation in a Knowledge Base comprising a variety of knowledge structures, e.g., ontologies, facts, rules, and constraints. The self-* objectives are specified with special policies associated with goals, special

[4] ARE was developed in a joint project by Lero, the Irish Software Engineering Research Center and ESA (European Space Agency)

situations, actions (eventually identified as system capabilities), metrics, etc.

ARE helped us build a knowledge representation model for the Science Cloud case study [VHBM13], which is at the right level of abstraction and relevance, i.e., carrying all the necessary details to represent the knowledge required to process self-adaptive behavior based on awareness capabilities.

2.1.2 Modeling and Programming

To deal with adaptation and move toward the actual implementation of the self-* properties identified in the previous section, the SCEL language [DLPT13], [DFLP11a] has been designed. It brings together programming abstractions to directly address aggregations (how different components interact to form ensembles and systems), behaviors (how components progress) and knowledge manipulation according to specific policies. SCEL specifications consist of cooperating components which, as shown in Figure 2 below, are equipped with an interface, a knowledge repository, a set of policies, and a process.

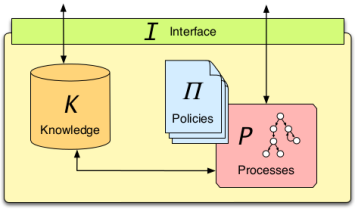


Figure 2: A SCEL component.

Behaviors describe how computations progress and are modeled as agents executing actions. *Interaction* is obtained by allowing components to access knowledge of components. *(Self-)Adaptation* is enabled by knowledge acquisition and is implemented through process manipulation. In this way, components can self-configure to adapt dynamically to changes in the environment, or initiate self-healing actions to deal with system malfunctions, or install self-optimizing behaviors.

Knowledge repositories provide the high-level primitives to manage pieces of information coming from different sources. Knowledge is represented through items containing either *application data* or *awareness data* with the latter providing information about the external environment (e.g. monitored sensor data) or about component status (e.g. its current location). This enables context- and self-awareness.

Interfaces are used to make available to other components selected parts of the knowledge of each component. An interface characterizes the component itself and can be queried to extract information about, e.g., the status, the offered services, or the execution environment. It can be seen as providing a set of *attributes*, i.e. names, acting as references to information stored in its knowledge repository.

Policies control and adapt the actions of the different components for guaranteeing accomplishment of specific tasks or satisfaction of specific properties. They regulate the interaction between the internal parts of a component (*interaction policy*) and with other components (*authorization predicate*).

Aggregations describe how different entities are brought together to form components and to construct the software architecture of ensembles. Components' composition and interaction are implemented by exploiting the attributes exposed in the interfaces. This form of semantics-based aggregation permits defining highly dynamical ensembles, that can be structured to dynamically adapt to changes in the environment or to evolving goals.

The language is equipped with an operational semantics that permits verification of formal properties of systems. Moreover SCEL program can rely on separate reasoning components that can be

invoked when decisions have to be taken. The reasoner is provided with information about the relevant knowledge SCEL program have have access to the programs receive in exchange informed suggestions about how to proceed.

To move toward implementation, jRESP[5], a JAVA runtime environment has been developed that provides an API that permits using in JAVA programs the SCEL's linguistic constructs for controlling the computation and interaction of autonomic components, and for defining the architecture of systems and ensembles. Its main objective is to be a faithful implementation of the SCEL programming abstractions, suitable for rapid prototyping and experimentation with the SCEL paradigm. The large use of design patterns greatly simplifies the integration of new features. These technologies simplify the interactions between heterogeneous network components and provide the basis on which different runtimes for SCEL programs can cooperate. It is worth noticing that the implementation of jRESP fully relies on the SCEL's formal semantics. This close correspondence enhances confidence on the behaviour of the jRESP implementation of SCEL programs, once the latter have been analysed through formal methods made possible by the formal operational semantics.

2.1.3 Verification and Validation

When dealing with complex autonomic systems one needs to face the problem of the development and of the validation of the models used for planning and for execution control. Indeed, while it is important for a large class of autonomic systems to integrate sensing and acting functionalities, controlled by deliberation mechanism (e.g. planning and execution control), the actual integration very often follows simple rules of thumb, which do not rely on any clear verification and validation approach.

Nevertheless, the autonomy requirement of these systems keeps rising, and they need a more flexible approach to handle the used resources. These systems are deployed for increasingly complex tasks; and it becomes more and more important to prove that they are safe, dependable, and correct. This is particularly true for rovers used in expensive and distant missions, such as Mars rovers [BdSIY13], that need to avoid equipment damage and minimize resource usage, but also for robots that have to interact regularly and in close contact with humans or other robots. Consequently, we think that it is becoming very common to require software integrators and developers to provide guarantees and formal proofs as certification.

Formal verification is an attractive alternative to traditional methods of testing and simulation that can be used to provide correctness guarantees. By formal verification we mean not just the traditional notion of program verification, where the correctness of code is at question. We more broadly mean design verification, where an abstract model of a system is checked for desired behavioural properties. Finding a bug in a design is more cost-effective than finding the manifestation of the design flow in the code. The ASCENS approach relies on the integration of two state-of-the-art technologies for verification and validation, namely D-Finder [BBNS09, BGL$^+$11] and SBIP [BBD$^+$12]. They are both based on BIP, a formal framework for building heterogeneous and complex component-based systems [BBS06]. Notably, thanks to the formal operational semantics of the SCEL language outlined in the previous section, BIP models can be obtained from static SCEL descriptions (i.e. involving only bounded creation/deletion of components and processes) by exploring a set of transformations rules.

For further details about the application of verification and validation techniques and corresponding tools the reader is referred to the ASCENS Joint Deliverable JD3.1. [Be13].

[5] jRESP website: `http://code.google.com/p/jresp/`

2.2 Running Self-Aware Systems

The "second wheel" representing the *runtime* or *online* phases comprise *monitoring*, *awareness* and *self-adaptation*. The focus of these phases are the observation of the system and the environment, reasoning on such observations and using the results of the analysis for adapting the system, that continuous being monitored. In addition, feedback is provided for offline activities, that will result in an evolving system through adaptation of the requirements, models or code.

2.2.1 Monitoring

Monitoring is the activity and the mechanism used at runtime to collect data for the purpose of awareness. Both individual components of an ensemble and the environment where they operate are monitored.

In the double-wheel life cycle, monitoring has a dual role. The usual primary objective is to provide information about the current state of the components and the environment to the awareness mechanism, which incorporates this information into the decision making process. Coupled with this is the second objective, to provide developer feedback about the behavior of the awareness mechanism, and check whether it is executing within the intended parameter domain.

One of the technical challenges to be faced is *dynamic coverage configuration*, where the awareness mechanism may require different information at different points. Monitoring should accommodate requests for information dynamically, rather than relying only on a statically configured description of what has to be monitored. It is also important to provide *monitoring cost awareness*, to make it possible to reason on the trade off between the cost of monitoring and the benefit of awareness, and *high monitoring coverage*, to accommodate the requirements of the awareness mechanism.

To support easy access to monitoring information in ASCENS, we have developed SPL [BBK+12], a formalism that makes it possible to express conditions on performance related observations in a compact manner. To collect the monitoring information from executing components, we use dynamic instrumentation in DiSL [MVZ+12]. In [BBH+12], we explain how the two technologies interact in the context of a performance aware component system.

2.2.2 Awareness

Awareness comprises the knowledge of the system and its environment as well as the reasoning mechanisms that an ensemble can employ at runtime. We divide the notion of awareness along three main dimensions: *expressivity* (Which parts of the system and environment are represented by the awareness mechanism and how detailed is this representation?), *quality* (How well do the conclusions of the awareness mechanism correspond to reality?) and *interface* (How is the awareness mechanism connected to the rest of the system?). We further classify expressivity as *scope* and *depth* which are closely connected to the SOTA (or GEM) models [HW11]: the scope represents the dimensions of the SOTA model that are contained in the awareness model whereas the depth corresponds to meta-information about the SOTA model contained in the awareness mechanism. For example, if one dimension of the SOTA model describes the location of a robot and this data is contained in the awareness model of the robot, the location is part of the model's scope. For physical robots this data has the property that it can only change continuously. This kind of knowledge is typically not part of the SOTA model; if it is explicitly represented in the awareness mechanism (e.g., in the form of logical axioms), it is comprised in the depth of the awareness model. More details can be found in [HW14].

To enable some kinds of problem solving and adaptation in complex domains, *deep awareness mechanisms* may be required, i.e., awareness mechanisms that contain meta-knowledge about the domain they describe. Deep models and reasoners can not only answer questions about the immediately

observable state of the system; since they also model underlying principles such as causality or physical properties they may, e.g., infer consequences of actions or diagnose likely causes of unexpected events.

However, an expressive awareness mechanism is not sufficient to successfully operate in open-ended environments: Designers usually cannot provide a complete specification of the conditions encountered by an autonomic system at runtime and all contingencies that may arise. To perform well in partially unknown environments and to allow flexible reactions in unforeseen situations, the awareness mechanism will have to adapt its internal models to the circumstances encountered at runtime. Therefore the awareness mechanism may often need to combine declarative reasoning with machine learning techniques to maintain the required quality of awareness during the system's lifetime.

The POEM language [Höl13] enables developers to specify deep logical and stochastic domain models that describe the expected behavior of the system's environment. System behaviors are specified as *partial programs* (also called, somewhat ambiguously, *strategies*), i.e., programs in which certain operations are left as non-deterministic choices for the run-time system. A strategy for resolving non-determinism is called a *completion*. Various techniques can be used to build completions: If precise models of the environment are available for certain situations, completions may be inferred logically or stochastically, and planning techniques can be used to find a long-term strategy. In cases where models cannot be provided, reinforcement learning techniques can instead be applied, and the ensemble can behave in a more reactive manner. POEM is based on the same mathematical foundations as KnowLang, therefore models developed using the Autonomy Requirements Engineering (ARE) approach (see Sect. 2.1.1) that do not make use of the advanced features of KnowLang can straightforwardly be translated into POEM.

The *Iliad*[6] implementation of POEM includes various built-in reasoners for computing completions of partial programs which can be coordinated using a blackboard system: (i) a theorem prover provides facilities for full first-order inference and also integrates special-purpose reasoners for, e.g., temporal and spatial reasoning; (ii) an ontology reasoner provides specialized reasoning about relationships between ontological concepts (iii) a HTN planner [NGT04] can compute long-term strategies if enough information about the ensemble's operating conditions is available; (iv) a hierarchical reinforcement learning subsystem can learn completions based on either a model of the system's environment or purely based on information gathered at run-time without explicit knowledge about the environment. The built-in blackboard greatly simplifies the integration of special-purpose reasoners, either at design time or even at run time.

Iliad is integrated as knowledge repository and reasoner in jRESP and can therefore be used as awareness engine for SCEL programs. We have also developed Hexameter[7], a portable implementation of the SCEL tuple-space primitives that can be used to directly connect *Iliad* to programs written in C/C++, e.g., the ARGoS swarm robotics simulator. *Iliad* can be used to build awareness mechanisms using a range of sophisticated reasoning and learning mechanisms. Models developed using the ARE approach can be executed using *Iliad* after they have been translated into POEM; once the full KnowLang reasoner becomes available it will be possible to use ARE models directly as awareness mechanisms.

2.2.3 Self-adaptation

Once components and ensembles have reached the awareness that there exist malfunctions, contingencies, or simply performance issues that require adaptation, some form of decision making should take place to evaluate the possibility for an adaptation action, and then such adaptation action must be

[6] `https://github.com/hoelzl/Iliad/`
[7] `https://github.com/hoelzl/Hexameter/`

eventually executed.

In ASCENS we distinguish between two main classes of adaptation actions:

- *Re-configuration*, aka weak self-adaptation, which implies modifying some of the control parameters of a component/ensemble, and possibly adding new functions/behaviors or modifying some of the existing ones.

- *Self-expression*, aka strong self-adaptation, which implies modifying the very structure of the component or ensemble, and in particular modifying the architecture by which adaptive feedback loops are organized around the component or ensemble.

From a different perspective, consider a component or an ensemble architected accordingly to one of the self-adapting patterns selected after the SOTA requirements modeling phase. Then, an action of re-configuration simply implies the specific autonomic feedback loop(s) (better, the autonomic managing components within) involved in such patterns to affect the parameters and behavior of the controlled components or ensembles, yet without affecting its own specific goals and structure. On the other hand, an action of self-expression implies that the autonomic managing components of a feedback loops recognize that its own very structure is no longer adequate to support the changed conditions, and re-shape itself into a new architectural pattern.

Now, given a component or an ensemble architected according to one of the self-adapting patterns via a SOTA model, re-configuration concerns the change of parameters without changing the structure of the feedback loops. Self-expression, in contrast, modifies the structure of the feedback loops themselves. To the best of our knowledge, ASCENS is the first approach in which both weak and strong forms of self-adaptation are put at work in a unique coherent framework.

2.3 Transitions between Design and Runtime Cycles

The two cycles of EDLC are complemented by transitions from design cycle to runtime cycle and vice versa. These transitions thus correspond to deployment and feedback activities. Further, together with the design and runtime cycles form an overall development cycle, providing the mechanisms to change architectural models and code on the basis of the runtime behaviour of the continuous evolving system. This way long term system evolution is encompassed, in which monitoring data observed at runtime are fed back to design cycle to provided basis for system redesign and redeployment.

2.3.1 Deployment

The deployment transition serves for preparing a service component application for runtime phase. This involves installing, configuring and launching the application. The deployment may also involve executable code generation and compilation/linking. In ASCENS, the deployment is addressed by service-component runtime frameworks (such as jDEECo [BGH$^+$13a] and jRESP). These frameworks allow for distributed execution of a service component application and provide their specific means of deployment.

2.3.2 Feedback

The feedback transition takes data collected by monitoring a running application back to the design phase to be analyzed and used for improving corresponding components. It connects the runtime monitoring with design. This connection is made possible by employing design methods that keep the traceability of design decisions to code artifacts and knowledge – e.g. Invariant Refinement Method

(IRM) [KBP+13a], which has been specifically developed for hierarchical design of a service component application). When used in conjunction with IRM, monitoring (a) observes the real functional and non-functional properties of components and situation in components environment, and (b) provides observed data to the design. At design time these observed data are compared to assumptions and conclusions captured by IRM. If a contradiction is detected, IRM is used to guide a developer to a component or ensemble which has to be adjusted or extended, e.g. to account for an unexpected situation encountered at runtime.

3 EDLC in the Context of the ASCENS Case Studies

Applying the Ensemble Development Life Cycle (EDLC) process, as defined in the previous section, in practice is the ultimate goal of the ASCENS approach. Most of the methods, techniques and tools were developed with a straight-forward practical deployment in mind. In the previous project years, separate approaches where individually tested on the project case studies. For example, some of the major concepts developed within the project, like the SOTA and SCEL approaches, have been constantly fine-tuned, taking into account the feedback from the pragmatic case studies deployment. Such a strategy made ASCENS case studies both, a source of inspiration for theoretical work and a playground for testing and improving the concepts. Finally as the concepts grew mature, the ASCENS case studies are playing a role of practical justification of the project theoretical and methodological work. The final result is a set of generic tools that can be applied in a wide application domain, ranging from robotics, cloud computing to e-mobility. The decisive outcome is to have runnig applications that exhibit advantages (e.g. self-awareness, knowledge richness, highly dynamic configuration, autonomous behavior) that can hardly be achieved with state-of-the-art technology. Last but not least it is possible to reason, validate and verify these features using ASCENS tools.

In this section the ensemble development life cycle (EDLC) is further detailed and exemplified on three ASCENS case studies: science cloud, e-mobility and swarm robotics. EDLC applied to each of the case study is presented through requirements analyses, modeling, programming and deployment. Both initial and run-time phases are taken into account. Only verification is left out of this report and is thoroughly reported in JD3.1.

Three completely different case studies from diverse application domains were used to test the ASCENS high-level tools. Applying the abstraction as the prime software engineering principle, a number of generic common features that characterize each application are extracted (e.g. existence of high number of individual entities with individual goals; existence of numerous collective goals that require dynamic grouping, need for knowledge, awareness and collective autonomous behavior, adaptation, robustness, etc). These features are used in requirement analyses to form knowledge bases and to build awareness and self* features of the system elements. For requirement analyses the SOTA methodology is applied to each of the case studies. In the swarm robotics scenario, a further prototyping tool (MESSI) was used to interface the SOTA approach and SCEL and jRESP approaches. The modeling phase uses the SCEL language to model and express dynamism in individual vs. collective behavior. Programming and deployment is done using jRESP and DEECo frameworks that practically map SCEL process algebra to Java programming language. Further ensemble modeling tool like HELENA is used within science cloud, whereas Argos is used as simulation framework for swarm robotics. The monitoring, Awareness and Self-Adaptation phases of the run time cycle are supported by custom tools close to the run-time framework (still under development). One example of a tool to reason about system awareness is the Iliad system, a Poem-based framework that allows monitoring of knowledge objects within running system. With the help of Iliad, a SOTA based adaptation specified with its goals and utilities, modeled in SCEL and implemented in jRESP, can be examined at run-time.

3.1 Cloud Computing

The cloud computing case study discussed in ASCENS centers on an autonomic cloud computing platform; or, in other words, a distributed software system which is able to execute applications in the presence of certain difficulties such as leaving and joining nodes, fluctuating load, and different requirements of applications to be satisfied.

The Cloud Computing case study centers on a platform as a service (PaaS) cloud solution which we call the Science Cloud Platform (SCP). This platform combines a standard PaaS cloud computing

infrastructure with peer-to-peer and voluntary computing, respectively; the full description of the cloud computing case study of ASCENS can be found in deliverable D7.3 [Ser13].

Such an infrastructure requires autonomic nodes which are (self-)aware of changes in load (either from cloud applications or from applications external to the cloud) and of the network structure (i.e. nodes coming and going) which requires self-healing properties (network resilience). Another issue is data redundancy in case nodes drop out of the system, which requires preparatory actions. Finally, executing applications in such an environment requires a fail-over solution, i.e. self-adaptation of the cloud to provide what we may call application execution resilience. It is not necessary in this context to prevent participation of partially centrally-controlled entities such as IaaS providers. In fact, parts of the SCP may run on IaaS solutions which enables it to spawn new virtual machines or shut them down again. Such additional functionality can be used to balance load or to conserve energy.

To sum up in one sentence, *the goal of the SCP is to deploy and run user-defined applications on the p2p-connected web of voluntarily provided machines which form the cloud.*

In the following, we introduce various ASCENS methods that have been applied to the cloud computing case study. We start with sections on requirements, modeling and programming: First, we discuss requirement analysis in section 3.1.1 followed by awareness patterns in section 3.1.2. We then continue with modeling in Helena in section 3.1.3, and finally programming in SCEL and SACPL in section 3.1.4. Afterwards, we discuss runtime concerns: First, in section 3.1.5, we discuss a collaborative approach for distributed task execution; finally, an excursion into the area of mobile cloud computing is presented in section 3.1.6.

3.1.1 Requirements Engineering with SOTA

The precise analysis of requirements is an important aspect in the early phase of system engineering [ST09]. Depending on the system under consideration, the domain must be mapped out in different ways. In open-ended and adaptive systems, a goal-oriented approach to requirements engineering, i.e. where requirements are modeled in terms of *goals* [MCY99], is the most beneficial.

Within ASCENS, the SOTA (State Of The Affairs) approach [AZ12] captures this way of thinking about a system, and goes one step further from simple goal-oriented systems in that it considers dynamic systems modeling with a particular focus on systems with self-* (self-star) properties, in particular self-adaptation. Within a system, this applies both on a local level (i.e., the component level) and more global levels (i.e., the ensemble level).

The SOTA approach proves useful to understand and model functional and non-functional requirements, expressing them into system specifications whose correctness can be checked. A complete definition of the requirements of a system-to-be implies identifying the dimensions of the SOTA space, and in particular defining the set of goals (with pre- and post-conditions, and possibly associated goal-specific utilities) and the global utilities for such systems, that is, the sets:

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

where

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

$$\mathbf{U} = \{U_1, U_2, \ldots, U_n\}$$

The SOTA specification expressed in terms of *Goals* and *Utilities* enables the designer to choose the most appropriate pattern from a catalogue ([Puv12]), so as to describe the system under study at best.

Depending on the type of utilities and/or goals in the specification of the scenario, a pattern is more suitable than another, as thoroughly discussed in [PPC+13] and [PCZ13].

It is important to note that self-* systems should not be simply engineered to achieve a certain state, but to *strive* to achieve a certain state. In SOTA, the entities of the systems (either simple ones or collections of entities) are seen placed in an n-dimensional space $\mathbf{S}$, where each of the n dimensions represents one axis on which the entities are placed. The position of the entity may change either based on its own actions or because the environment changes. Moving within S thus corresponds to the evolution of the system.

The different dimensions in S within the cloud case study relate to the fitness of each node in itself, and to keeping to the requirements of applications (SLAs) executed by ensembles. Thus, dimensions for each component include

- CPU load, i.e. processor utilization on nodes (in an ensemble: of the cloud in general)
- Available main memory, i.e. utilization of the available RAM
- Executed applications, i.e. apps running on a certain node (in the cloud: and their proper distribution)
- Energy consumption, i.e. the drain on local energy resources which depends on machine parameters

A reputation score can be added which negotiates between energy usage and application execution. Application dimensions (given their SLAs are fulfilled) are

- Response time (of an application given a user request)
- Redundancy of data (how many copies are available)
- Cost of execution (for example in terms of energy, or of actual renting cost)

As basic entities, the cloud case study contains nodes (service components) and collections of nodes which aim to keep one application running (service component ensembles). We must also identify the *goals* and *utilities* for these entities.

For a node, basic goals are:

- not to degrade in speed (i.e., keeping the local CPU load on an acceptable level)
- as well as not exceeding its main memory
- but, on the other hand, keeping energy consumption in check.

How to deal with the trade-off between application execution and energy consumption is an interesting question which can be solved in different ways, for example by using reputation scores. On the ensemble level, key goals are:

- keeping applications running, that is always having one or more nodes available which execute an application
- keeping enough backups of both application code and data to survive contingency situations
- fulfilling application requirements, or SLAs.

Finally, utilities can constrain how a goal is achieved, that is by confining its execution path or trajectory. A utility related to load might restrict, for example, a load above 90%, which means that components need to avoid the area with these values in the state space S. On an ensemble level, we may express the wish to avoid idling machines, that is if all applications are executing in a stable way to shut down single nodes, which can be expressed by a utility which codifies the maximum percentage of idle nodes.

Balancing the load in a node can be defined as follows:

$$Gnode_i = \quad \{ \quad \begin{array}{l} G^{pre} = 25\% < load\_of\_work < 75\%; \\ G^{post} = load\_of\_work > 50\%; \\ U = energy\_level > 0 \end{array} \quad \}$$

The main benefit of using SOTA is understanding the functional and adaptation requirements of the system. For design, we can move forward using adaptation patterns to identify the possible architectures for the system, which is discussed in the next section.

3.1.2 Adaptation Patterns applied to Science Cloud

A common approach to understanding, categorizing, and designing IT systems is the use of patterns, i.e. descriptions of characteristics which have proven to be beneficial for the implementation of a system. Within ASCENS, a catalog of architectural design patterns has been developed [CPZ11] which are intended to be used to build adaptive components and systems. In the life cycle, this work is relevant on the border between requirements engineering and modeling.

The design patterns have been studied with regard to the cloud case study [PF13]. In this section, we will discuss two patterns which have been used in the cloud.

Firstly, we need to discuss individual cloud nodes (which we call SCPis, for Science Cloud Platform instances). In this regard, the *Proactive Service Component pattern* [PF13] best captures the behavior of such a node. This pattern enables the SCPi, which is a *Service Component* (SC) in the terms of ASCENS and the adaptation pattern itself, to have an internal feedback loop, or, in other words, implicitly contain an *Autonomic Manager* (AM) which is responsible for driving the adaptation through this feedback loop. These kinds of components are used because nodes in the cloud are goal-oriented in nature and actively try to adapt their behavior, even without an external call (e.g. for saving energy). A visualization of such a component is shown in Figure 3.

In the cloud, the sensor and effector are used as follows:

- *Sensor*: the sensor part of a cloud node uses OS-specific functions to determine the CPU load, available memory, network utilization, and so on.
- *Effector*: If an IaaS system like the Zimory Cloud is available, a cloud node may use the API of such a platform as its effector, creating new virtual machines or shutting them down.

Input and output refer to user interactions:

- *Input/Output*: Cloud nodes run applications, and as such they must handle user input (in the form of web requests) to these applications, and sending the result of the request back to the user. This interface is HTTP-based.

The control and emitter ports are used for ensemble adaptation (see below).

By using the proactive service component pattern, individual SCP nodes are self-aware and able to self-adapt, each following the goal of achieving best performance for deployed apps while saving energy. The internal feedback loop created through the AM part of the node is used for checking these conditions and adapting properly.

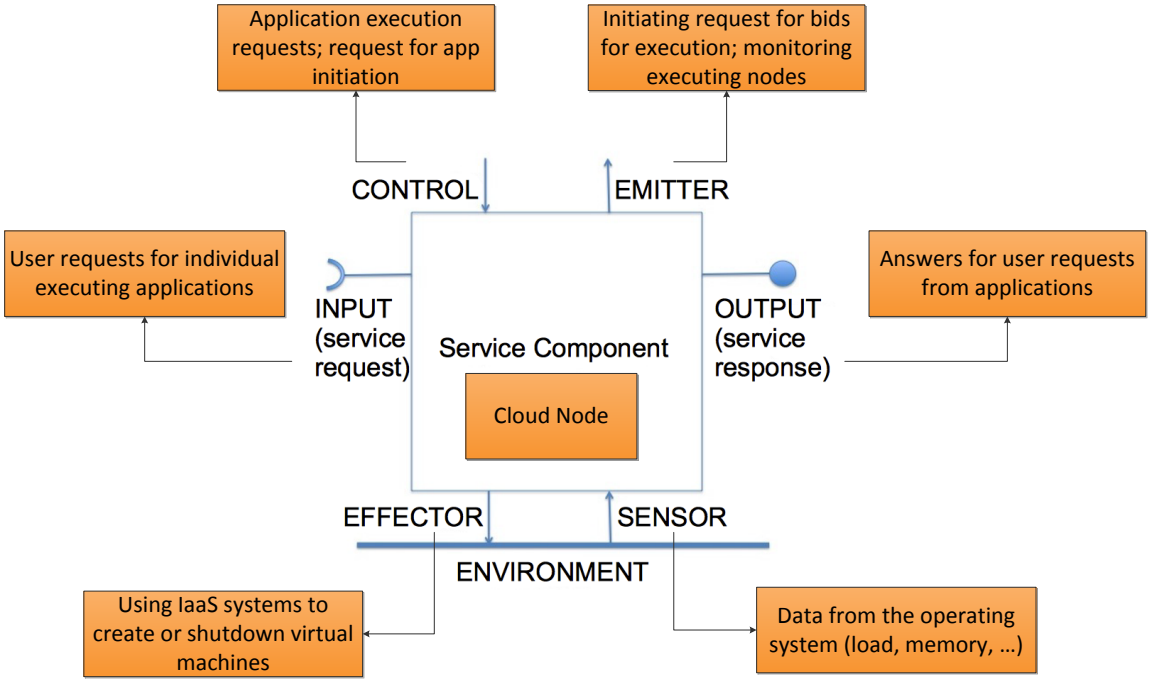


Figure 3: Proactive Service Component

Furthermore, multiple nodes work together to execute applications. On this level, the *P2P Negotiation Service Components Ensemble Pattern* [PF13] is a fitting description of this behavior, since each node (potentially) communicates with every other node for adaptation, there is no central coordinator, and each node follows a goal (which in this case is the same for each node, though with different data depending on deployed apps). The use of this pattern is also possible because the components that form the ensemble are proactive and need to communicate with others to propagate adaptation. This is done, as indicated above, through the control and emitter interfaces of the service component:

- *Emitter*: service components in a deploy or initiator role (see also the modeling section) emit requests for initiation and execution to other nodes. Furthermore, an executor lets other nodes know via its emit port when it is no longer able to execute an application.
- *Control* Via the control interface, the above requests are received and acted upon.

Using this pattern, multiple SCP nodes work together: For each application, one ensemble consisting of a subset of the overall cloud nodes is formed which is then responsible for executing the application (which includes deployment, finding an executor, executing, and monitoring). We call such an ensemble an SCPe (Science Cloud Platform ensemble).

Obviously, there are also other ways in which a cloud can be organized. In [PF13], the applicability of the *Centralized AM Service Components Ensemble Pattern* was discussed as well. This pattern proposes a completely different setup which does not use a peer-to-peer organization but instead uses a centralized autonomic manager. Dynamically adapting the cloud to such a structure might be advisable

in the case of a partial blackout of the cloud, that is, a large percentage of the cloud goes down. If only a few nodes remain, switching to a centralized mode in which one AM coordinates many individual nodes (which give up their own adaptivity mechanisms for the time being) might prove to be more effective. Nevertheless, this pattern can only be applied for the time that its context of applicability is the same as in the observed case. When the context changes again, the pattern has to be changed as well.

3.1.3 Ensembles Level Modeling with Helena

Modeling the behavior of the individual components and the ensembles which implement the cloud functionality is challenging due to the complexity and dynamics of the participating ensembles. In ASCENS, existing techniques such as component-based software engineering ([Szy02, RRMP08]) have thus been augmented with features that focus on the particular characteristics of ensembles. Among these are the fact that ensembles are dynamically formed on demand, realizing collective, goal-oriented behavior through communication between the individual participants; furthermore, multiple ensembles may run concurrently using the same basic resources, but dealing with different tasks on a higher level. To be able to model these issues on a first-class basis, the *Helena* approach [HK14] has been developed, which uses a UML-like notation for collaborations founded on a rigorous formal semantics. This work may be used in the modeling section of the EDLC.

A particular property of ensembles is the fact that although the platform on which ensembles run may itself be plain component-based, each component can take part in different ensembles and in the course of doing so take up different, ensemble-specific *roles* [GSR96]. A service component may play different roles at the same time, both in one ensemble and in different, concurrently running ensembles; it may also dynamically change its role(s) in order to adapt to new situations.

The Helena approach is centered on this notion of roles and the collaboration of roles in ensembles for pursuing the ensemble goal. In the present case study, there may be multiple such ensembles; one for each of the applications which are executed within the cloud. Each ensemble has the goal of deploying the application, finding an execution target node, executing, and finally monitoring the application execution. This is illustrated in Figure 4.

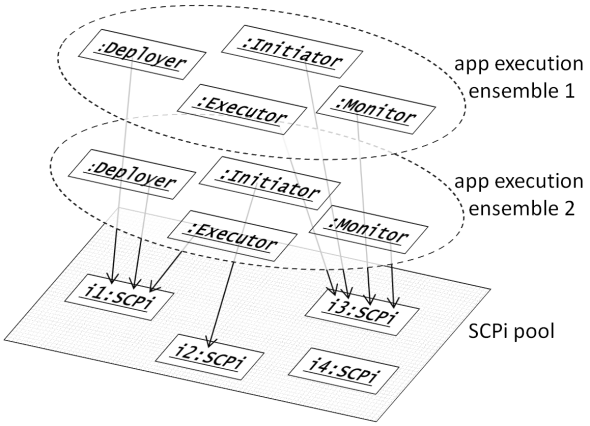


Figure 4: Ensembles in the *Helena* approach

The first or basic level (on the bottom of the figure) shows the pool of all SCPi nodes which are, in principle, able to provide resources to the cloud. In the figure, these are the four nodes labeled i1 to i4, which may be physical or virtual machines on which instances of the science cloud platform (SCPis) are running. Each of these may participate in ensembles for executing an application. As indicated

in the figure, executing an application requires different responsibilities taken up by different roles in the ensemble. These are the deployer (node from which the application originates), the initiator (leading the search for an execution node), the actual executor, and a monitor which keeps tab on the executor. As an example, the figure shows two different ensembles, each executing one application, where nodes concurrently play different roles or do not participate at all.

Ongoing research in Helena currently focuses on the description of the behavior of each role as well as on the behavior on the ensemble level. These descriptions are given a rigorous formal foundation, which can then be exploited for ensuring that the ensemble behavior actually reaches the desired goal. We believe that the analysis of ensembles of collaborating roles can be beneficial to developers due to the reduction of the complexity of the models, since the combination of all roles within one service component must only be integrated into a component-based architecture in the following implementation phase. This is discussed in the next section, where a language is presented to which a systematic transition from Helena is currently being investigated.

3.1.4 Modeling the high-load Scenario with SCEL and SACPL

ASCENS has been studying linguistic primitives suitable for the autonomic computing paradigm, and has developed the language SCEL (Software Component Ensemble Language) [NFLP13, DLPT13] which is geared towards describing autonomic systems, taking into consideration the behaviors, knowledge, and aggregations involved, based on specified policies. SCEL in particular supports programming context-awareness, self-awareness, adaptation and ensemble-wide interactions — and may be used in the modeling/programming section of the EDLC.

In the following, we discuss the application of SCEL to the service components of the cloud case study. The concept of a service component – or autonomic component – lies at the heart of SCEL. This concept directly matches the notion of an SCPi, i.e. an individual node in the science cloud. Furthermore, the notion of an ensemble in SCEL matches the notion of an SCPe, since both are based on components' attributes, which in the science cloud usually take the form of participation in the management of a cloud application.

As an example, we consider here the SCEL implementation for a situation in the cloud where a node is overloaded, i.e. the CPU load exceeds a certain threshold and an application needs to be moved to a different node. This scenario includes the use of an IaaS solution, that is we include the ability to spawn a new virtual machine and moving the application there.

The full SCEL specification for the scenario of high load and moving an element to a newly created VM can be found in [DLPT13]. We will outline the general idea of the behavior here. The SCPi where the application is running initially is the following SCEL component:

$$\mathcal{I}[\mathcal{K}, \Pi, (AM[ME])]$$

The interface $\mathcal{I}$ of the component encapsulates the remaining three elements. $\mathcal{K}$ represents the knowledge of the SCPi, which includes attributes relevant for adaptation. Π is the policy the component follows, which in this case is specified in SACPL, the SCEL Access Control Policy Language [DLPT13], discussed below. $AM[ME]$ is the (controlled) composition of processes AM and ME running in the component.

As we have seen in the section on adaptation patterns, an SCPi follows the proactive service component pattern. This means it contains, as in a SCEL component, internal knowledge and goals. In the above definition, the main work of the node, including the application logic, is performed in the Service Component (SC) which here is called Managed Element (ME). The component also contains its own, implicit, Adaptation Manager (AM), which specifies actions for adaptation (in particular, spawning a new machine). In the considered scenario, the managed element cyclically reads (by

means of action **qry**) a local datum, e.g. a key, elaborates (by means of function $f(\cdot)$) it and sends (by means of action **put**) the result to the component m:

$$ME \triangleq \mathbf{qry}(\text{“}key\text{”}, ?\, x)@\mathsf{self}.\mathbf{put}(\text{“}key\text{”}, f(x))@m.ME$$

Instead, the adaptation manager just spawns a new machine (by means of action **new**):

$$AM \triangleq \mathbf{new}(\mathcal{J}, \mathcal{K}, \Pi_J, AM[\,ME\,])$$

The actual adaptation logic (i.e., when to adapt) is dealt with using the policy Π. The component's interface $\mathcal{I}$ exposes the attribute *CPULoad*, whose value (i.e., a percentage of load) is a context information sensed by the component from the underlying infrastructure. The policy Π detects when the attribute value is over a given threshold (e.g., 80%) and triggers the autonomic manager. More specifically, the policy says that the main application logic, which is part of ME, may only be performed as long as *CPULoad* is less than the threshold, while the spawning of a new machine (realized by means of an action **new** in AM) may not be performed until *CPULoad* is greater than the threshold.

In particular, the policy Π in force at the component results from the composition, by means of the p-o (permit override) operator, of the following policies:

$\langle deny$; target: { } $\rangle$ * `deny all` *

$\langle permit$; target: { equal(subject.id,n) and
equal(object.id,n) and
equal(action,**qry**) and
less-or-equal-than(subject.CPUload,$threshold$) } $\rangle$ * `permit local` **qry** *

$\langle permit$; target: { equal(subject.id,n) and
equal(object.id,m) and
equal(action,**put**) } $\rangle$ * `permit remote` **put** *

$\langle permit$; target: { equal(action,**new**) and
greater-than(subject.CPUload,$threshold$) } $\rangle$ * `enable` **new** *

Basically, Π says that

- action **qry** may only be performed until subject.CPUload is less than, or equal to, $threshold$;
- action **put** may always be performed, leaving the value of subject.CPUload out of consideration;
- action **new** may not be performed until subject.CPUload is not greater than $threshold$.;
- all other actions that differs from those above are denied.

The rationale underlying this policy is that a **qry** may be computationally heavy (because it requires examination of the repository), while a **put** is a light operation (because it only requires addition of an item to the repository). Of course, different choices are possible.

An interesting problem in this context is that Π, ME and AM in a dynamically created VM are the same as those within the corresponding source node of the science cloud; however the application logic which is part of ME may only be executed on one machine at a time (since we assume that the application is a singleton). To ensure such behavior, multiple options have been explored with different power of expression. First, it is possible to add a new attribute to the component which keeps track of its execution status; AM is thus modified to properly set such an attribute. Second, the policy Π can be extended to include obligations that are actions executed as part of a node switch to take care

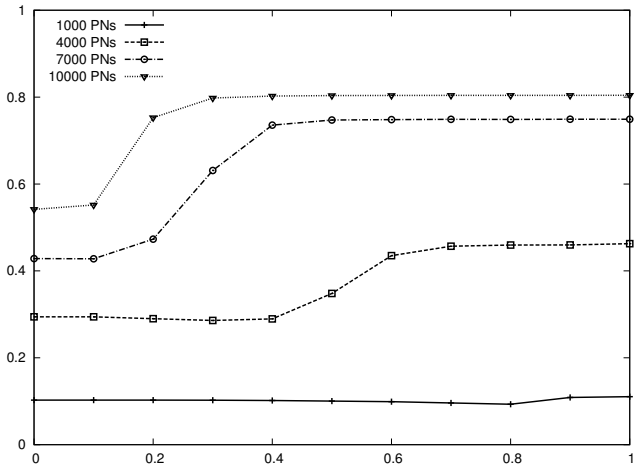


Figure 5: Performance with varying degrees of cooperation.

of dealing with the execution status attribute (in place of AM). Finally, it is possible to use several policies instead of a single one, and dynamically switch between policies on an adaptation by means of a sort of automata where states are policies and state transitions represent adaptivity events (expressed as policy targets). The details of these three options are discussed in [DLPT13].

To summarize, the above description has shown the use of SCEL and a policy language, SACPL, to model a scenario within the science cloud where high load of a node leads to the spawning of a new virtual machine with an additional SCPi which can take over the application logic. An implementation of these abstract descriptions can be done in Java (as discussed in the following chapter) or more directly in jRESP [DLPT13], which is currently work in progress.

3.1.5 A Cooperative Approach for Distributed Task Execution in Autonomic Clouds

In [ALLS13] we implemented a prototypical simulator inspired on the SCIENCE CLOUD, and have performed a series of simulation-based experiments to validate cooperative approaches. We considered different degrees of cooperation among PNs. A *selfish* cloud node sends requests for remote task execution when local resources are not sufficient, but rejects all external requests. Instead, a *volunteer* node always accepts external requests, when locally available resources are sufficient. In between the two opposite strategies, there is a partial volunteering scheme for which a node may decide to accept or reject a task execution request, depending on a function that takes depends on the willingness of the node to collaborate and its willingness to satisfy its own requests. The experimental evaluation is supported by DEUS [DEU] a general-purpose, discrete event simulator, that has been successfully applied for the modeling and simulation of various other complex systems. Our evaluation exploits real workload data from the Google Cluster dataset [Hel10] and provides estimations of the performance of various cooperation strategies for different SCIENCE CLOUD configurations showing that collaborative strategies tend to perform better than selfish ones, in particular for large number of nodes. As an illustrative example, Figure 5 shows how increasing the degree of volunteering (horizontal axis) and the number of nodes (the various curves) affect performance (vertical axis) in terms of the percentage of taks executed successfully (i.e. QoS respected).

This work is an example of how to use a simulation approach to test-drive the runtime circle of the Ensemble Development Life Cycle (EDLC).

3.1.6 Mobile Cloud Computing with DEECo

An interesting aspect of the case study is the fact that the individual nodes can be personal computers. As such, the concept also includes mobile nodes: laptops, tablets, or even smartphones. Mobile devices have some noteworthy properties in addition to standard nodes. They are devices (a) whose neighbors – in the sense of network proximity – may change, (b) whose battery capacity is limited, and (c) whose computing capacity may be (severely) limited as well.

Applications running on top of the science cloud may want to take those properties into consideration. In fact, we can imagine that applications intended to run on mobile devices be effectively split into two components, or smaller applications, communicating with one another. In one scenario, they may both run on one SCPi — if the node is powerful enough and access to power is not an issue; in another, they may be split between two SCPis, one on a mobile node (which handles UI) and another on a stationary node (which handles the computationally extensive background work). In order to keep the user interface responsive, the network latency between the two nodes may not exceed a certain threshold, which becomes problematic in the presence of (physical) node mobility.

This scenario has been investigated within ASCENS [BBHK13]. The envisioned method for this case uses a specialized adaptation architecture which, through two components, takes care of the planning and monitoring involved. This architecture spans the three activities in the runtime circle of the EDLC.

The first component involved is the *monitor*, which works within an application and can operate in one of two modes:

- *Running mode*. In running mode, the monitor executes as part of a running application, i.e. it reflects the actual deployment. The monitor gathers data about the current node, which includes the performance and battery life. This non-functional properties data (*NFPData*) is used by the planner (see below) to decide on adaptation.

- *Mock mode*. A monitor may also be detached from its application and spawned on a different node where it runs in mock mode, testing the performance of the node with the performance model of the application (*MonitorDef*) in mind, but without actually moving the whole application. Again, NFPData is generated which can be used by the planner.

The second component is the *planner*. The planner provides the SCPi with the MonitorDefs for the monitors involved, which the SCPi can distribute to interesting nodes for gathering NFPData. Based on information about the application, which are included in a deployment plan, the planner is able to restrict which nodes are interesting; for example, this may include nodes which are a limit of two hops away. Based on the information in the NFPData from affected nodes, the planner instructs the underlying SCPi(s) to deploy the applications appropriately given the data.

A particular advantage of the monitor approach with mock modes is the availability of real data: The monitor deployed on remote nodes is able to report, based on its MonitorDef, precisely those measurements which are relevant for the application. As usual, the nodes which may take part in the execution of an application form an ensemble with the specific task to figure out the best configuration for all entities involved.

Figure 6 shows a simplified definition of such an ensemble.

All in all, the adaptation architecture based on planners and (mock) monitors allows for a very flexible awareness of the network environment. While this approach is useful for all kinds of nodes the SCP may run on, it is particularly helpful in the presence of mobile nodes.

```
1 ensemble PlannerToDevice:
2   coordinator: Planner
3   member: Device
4   membership: HopDistance(Planner.device, Device) ≤ 2
5   knowledge exchange:
6     Device.monitorDef[Planner.app] := Planner.monitorDef
7   scheduling: periodic( 15s )
```

Figure 6: Ensemble Definition

3.2 e-Mobility

In this section, we explain how we exploited EDLC in the context of the e-Mobility scenario. The e-Mobility scenario focuses on avoiding contingency situation in an open-ended systems of interacting electric vehicles. Such a scenario is highly dynamic. This stems mostly from the fact that it includes unforeseeable human user actions which influence the availability of travel resources.

Technically, we assume in the case-study that travels are initiated by personal activities. A journey is thus defined as a sequence of trips, with each trip being initiated by a single activity. Trips may consist of multiple stages. A stage can be executed in different travel modes such as walking mode or driving mode. For example, consider a user that leaves for work in the morning. Work is the activity that initiates travel. The first trip contains a walking stage from home to the vehicle's parking lot, a driving stage from the parking lot at home to the one at work and lastly a walking stage to the office. The working time at the office is considered to be the activity duration. Throughout that time the vehicle is parked at the car park. If it has access to a charging station, it may recharge. After work the user continues his journey. The number of consecutive trips follows from the number of activities.

In this scenario the main components are the user, the electric vehicle, the parking lot and charging station. Parking lot and charging station are commonly referred to as infrastructure components. Component temporarily form ensembles. These ensembles include (i) collection of charging stations, (ii) collection of parking lots, (iii) collection of users and electric vehicles and (iv) collection of at least one user, one electric vehicle and one infrastructure component, etc.

Throughout runtime, contingency situations may occur. Components and ensembles require self-adaptive actions to resolve these situations. Examples of contingency situations that need to be resolved by the electric vehicle component include (i) unavailability of a reserved parking lot, (ii) unavailability of a reserved charging station, (iii) falling below minimum battery energy level and (iv) missing a scheduled arrival time. Examples of contingency situations that need to be handled by the parking lot or charging station component include (i) early or late arrival of a vehicle at a parking lot or charging station, (ii) early or late departure of a vehicle from a parking lot or charging station, (iii) missed initiation of a scheduled charging action and (iv) deviation from the expected power profile during charging.

3.2.1 Applying EDLC to e-Mobility – Big Picture

In the spirit of EDLC, we have employed several interrelated methods developed in ASCENS to address the application life cycle of the e-Mobility. In particular, we start with the specification of requirements and their reflection in the operation space of the system and system's self-awareness (described in Section 3.2.2). Following the requirements specification, we focus on the high-level architectural design, both in terms of adaptation patterns (Section 3.2.3) and in goals decomposable into individual components and ensembles (Section 3.2.4). Next, emphasis is put on low-level design

of component activities. For that, we employ the SCEL language, which is specifically intended for ensemble-based description of communication and coordination-oriented concerns (Section 3.2.5). Along with SCEL we also employ (Section 3.2.6) the SCLP (soft-constraints logic programming), which provides a natural way of describing optimization related tasks, which are very frequent in self-adaptive systems. The final step is the implementation of components and the deployment of the system (Section 3.2.7), for which we use a dedicated ensemble-based component model and component runtime (called DEECo).

3.2.2 Requirements Engineering with SOTA

The activity of requirements engineering for self-adaptive systems in SOTA requires identifying the dimensions of the SOTA space, which means modeling the relevant information that the different components and ensembles of a system have to collect to become aware of their location in such space. This is necessary to recognize whether they are behaving correctly and to adapt their actions whenever necessary.

In e-mobility, the space $\mathbf{S}$ clearly includes the spatial dimensions related to the street map, but also dimensions related to the current traffic conditions, the battery conditions, and in general any physical or virtual dimension that may affect the behavior of vehicles. As a vehicle moves along some road, its position in the SOTA space changes accordingly, obviously w.r.t. the dimension representing the spatial location but also w.r.t. the dimension representing the traffic and battery conditions.

Once the SOTA space is defined, a goal in SOTA can be expressed in terms of a specific state of the affairs to aim for, that is, a specific point or a specific area in $\mathbf{S}$ which the component or ensemble should try to reach in its evolution, in spite of external contingencies that can move the trajectory farther from the goal. For instance, a goal for a vehicle could imply reaching a position in the SOTA space that, for the dimensions representing the spatial location, trivially represents the final destination and for the dimension representing the battery condition may represent a charging level ensuring safe return. That is, if the location to be reached has coordinate *(x,y)* and we know that the place need to be reached in a time no more than T:

$$G_{car_i} = \Big\{ G^{pre} = location(t,s); G^{post} = location(x,y); \\ U = battery\_level > 10\% \bigcup time\_for\_reaching < T \Big\}$$

In general, a goal is not necessarily always active. That is, a goal has to be defined also in terms of the preconditions that activate it. In addition, a goal may impose constraints on the trajectory to be followed while trying to achieve, e.g., a car may wish to reach a destination while avoiding motorway.

3.2.3 From SOTA to High-Level Design with Adaptation Patterns

The SOTA modeling approach is very useful to understand and model the functional and adaptation requirements, and to check the correctness of such specifications (as described in [AZ12]). However, when a designer considers the actual design of the system, it is important to identify which architectural schemes need to be chosen for the individual components and ensembles.

To this end, in previous work [CPZ11], we defined a taxonomy of architectural patterns for adaptive components and ensembles of components. This taxonomy has the twofold goal of enabling reuse of existing experiences and providing useful suggestions to a designer on selecting the most suitable patterns to support adaptability.

In a coordinated system for e-mobility, the e-vehicles of a car-sharing company may all share the same basic adaptation goals, thus making it suitable to model them as simple components all sharing

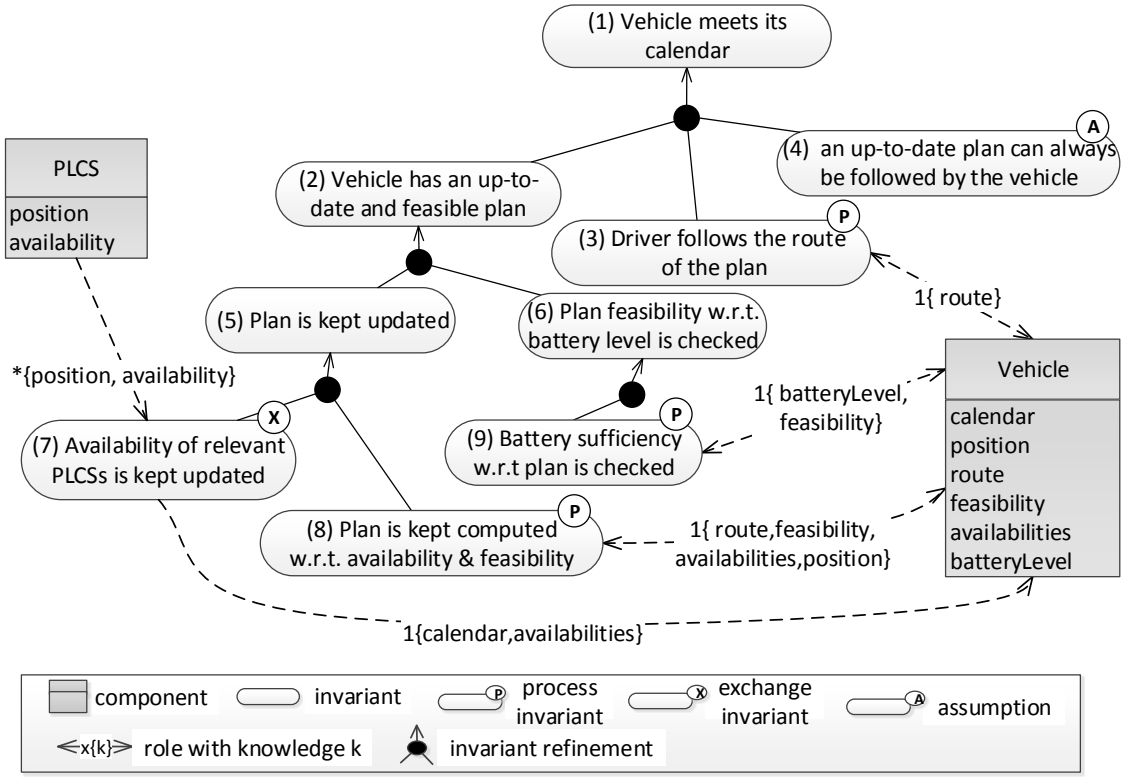


Figure 7: E-Mobility system level graph – IRM method.

the same class of external controller. Also, at the level of the fleet of e-vehicles, the presence of a single stakeholder makes it possible to exploit a pattern of an ensemble with a global control loop to orchestrate the overall behavior of the fleet.

To do that, an adaptation pattern that permits components to share the adaptation mechanism and goals is necessary and it is the "P2P Negotiation Service Components Ensemble Pattern". This pattern has a decentralised and shared control of adaptation and permits to the different components of the system to coordinate each other in order to manage adaptation (see [Puv12] for further details).

3.2.4 High-Level Design – Architecture

In order to guide the design of an ensemble-based system from high-level strategic goals, requirements and patterns (described by SOTA) to their low-level realization in terms of system architecture (components and ensembles) we use the *Invariant Refinement Method* (IRM) [KBP+13b].

The main idea of IRM is to capture the high-level system goals and requirements in terms of interaction *invariants*. In compliance to SOTA's notion of "striving to achieve", invariants describe the desired state of the system-to-be at every time instant. In general, invariants are to be maintained by the coordination of the different system components. At the design stage, by *component* we refer to a participant or actor of the system-to-be. A special type of invariant, called *assumption*, describes a condition that is expected to hold about the environment; an assumption is not intended to be maintained explicitly by the system-to-be.

As a design decision, identified top-level invariants are decomposed into more concrete sub-invariants forming a decomposition graph (Figure 7). The decomposition is essentially a refinement,

where the composition of the children exhibits all the behavior expected from the parent and (potentially) some more. By this decomposition, we strive to get to the level of abstraction where the (leaf) invariants represent detailed design of the particular system constituents components, component processes, and ensembles. Two special types of invariants, namely the *process invariants* (denoted by "P") and *exchange invariants* (denoted by "X"), are used to model the low-level component computation (processes) and interaction (ensembles), respectively.

A possible system-level graph corresponding to the simplified e-Mobility scenario is depicted in Figure 7. In this case, the IRM design mainly captures the necessity to keep the vehicles plan updated (invariant (5)) and to check whether the current plan remains feasible with respect to measured battery level (invariant (6)). The identified leaf invariants are easily mappable to component activities, which are further formally captured by SCEL or SCLP.

3.2.5 Modeling Computational Activities with SCEL

To complement the high-level, architectural design, we have proposed specific linguistic and programming abstractions aiming at dealing with the challenges posed to language designers by massively distributed and highly dynamic systems. Our starting points have been the notions of *autonomic components* (ACs) and *autonomic-component ensembles* (ACEs) that are used to structure systems into independent and distributed building blocks that interact and adapt in different ways. Based on the notions of ACs and ACEs, we have introduced a number of specific abstractions and linguistic constructs that permit building up ACs, defining ACEs and programming their behaviors and interactions. The proposed abstractions are the basis of SCEL (Software Component Ensemble Language) [DLPT13, DFLP11b].

ACs are entities with dedicated knowledge units and resources that can cooperate while playing different roles. Each AC is equipped with an *interface*, consisting of a set of *attributes*, such as provided functionalities, spatial coordinates, group memberships, trust level, response time, etc. Attributes are used by the ACs to dynamically organize themselves into ACEs.

Indeed, one of the main novelties of SCEL is the way sets of partners are selected for interaction and thus how ensembles are formed. Individual ACs not only can single out communication partners by using their identities, but they can also select partners by exploiting the attributes in the interfaces of the individual ACs. Predicates over such attributes are used to specify the targets of communication actions, thus providing a sort of *attribute-based* communication. In this way, the formation rule of ACEs is endogenous to ACs: members of an ensemble are connected by the interdependency relations defined through predicates.

Starting from the IRM model presented in Figure 7, we can identify two kinds of SCEL components, namely PLCSs and Vehicles:

$$\mathcal{I}_{plcs1}[\mathcal{K}_{plcs1}, \Pi_{plcs1}, P_{plcs1}] \parallel \mathcal{I}_{plcs2}[\mathcal{K}_{plcs2}, \Pi_{plcs2}, P_{plcs2}] \parallel \ldots$$
$$\parallel \mathcal{I}_{vehicle1}[\mathcal{K}_{vehicle1}, \Pi_{vehicle1}, P_{vehicle1}] \parallel \mathcal{I}_{vehicle2}[\mathcal{K}_{vehicle2}, \Pi_{vehicle2}, P_{vehicle2}] \parallel \ldots$$

A PLCS identifies a *parking lot/charging station* and is characterized by a *position*, its *availability* and, possibly, a *chargingStation*. These are the attributes that are exposed in the component interface and that respectively identify the location of the PLCS, the number of available slots in the area and the presence of a charging station.

$$\mathcal{I}_{plcs\,j} \triangleq \{(id, plcs_j), (type, \text{“}PLCS\text{”}), (position, (x_j, y_j)), (availability, n), (chargingStation, b)\}$$

As expected, Vehicle components identify cars involved in the scenario and will expose in their interface a set of attributes describing the state of the component (*position*, *batteryLevel*,...).

$$\mathcal{I}_{vehicle\,i} \triangleq \{(id, vehicle_i), (type, \text{“}Vehicle\text{”}), (position, (x_i, y_i)), (batteryLevel, \text{“}high\text{”}), \ldots\}$$

Attributes of both PLCSs and Vehicles are obtained as the projection on the interface of the local knowledge of each component.

The user associated to a vehicle is modeled by a process that, according to the local Vehicle interface, will interact with PLCSs in order to identify the next stop in the travel. This task is largely simplified thanks to the use of attribute based communication. Indeed, if *poi* is the next *point-of-interest* to visit in the travel, then the next PLCS to use can be identified by sending a *reservation request* to all the PLCSs components that are close to *poi* up-to a given *walking distance* and that can be reached with the current battery level.

$$P_{vehicle\,i} \triangleq \ldots\ \mathbf{get}(\text{“}poi\text{”}, j, ?x_{poi}, ?y_{poi})@\mathsf{self}\,.\ \mathbf{put}(\text{“}reservationRequest\text{”}, resData, \mathsf{self})@\mathsf{P}_{plcs}\,.\ \ldots$$

where P_{plcs} stands for predicate

$$\begin{array}{l} type = \text{“}PLCS\text{”} \\ \wedge\ \mathrm{distance}((x_{poi}, y_{poi}), position) \leq \mathrm{walkingDistance} \\ \wedge\ \mathrm{isReachable}(position, \mathsf{this}.batteryLevel) \end{array}$$

Keyword this indicates that $batteryLevel$ refers to an attribute specified in the interface of the vehicle; all other attributes (i.e., $position$ and $type$) refer to the PLCS components.

However, when the battery level of a vehicle decreases under a given threshold, the actual behaviour can be adapted so to force the reservation of a PLCS that can be used to recharge the battery and then continue with planned trip. In this case, the predicate used as target of the **put** action for sending the reservation request becomes $\mathsf{P}_{plcs} \wedge chargingStation = \mathrm{true}$.

We refer to Deliverable D1.3 for a more detailed account of an e-Mobility scenario modelled in SCEL.

3.2.6 Adaptation via Soft-Constraints Solving and Optimization

As a complement to SCEL specifically targeting intuitive specification of optimization problems that frequently appear in self-adaptive systems, we have used our approach on Soft Constraint Logic Programming.

Constraint logic programming (CLP) [JL87] extends logic programming (LP) by embedding constraints in it. They represent a limitation of the possible combinations of the values of some variables modeling a problem. For example, if we consider three variables $\{X, Y, Z\}$ which can take values in $D = \{red, yellow, blue\}$, we can have a constraint on each pair of them stating that they must take a different value: so for each assignment of colours they say if it is allowed or not.

A further extension of CLP has been proposed in [BMR01] to also handle soft constraints, which roughly speaking are constraints that rather than returning a boolean value yield more informative values such as a level of satisfiability or a cost. Technically, this is done by adding a structure modeling these levels, which is represented by a semiring, that is, a set with two operations: one is used to generate an ordering over the levels and another one is used to define how two levels can be combined and which level is the result of such a combination. So for example, in the case above, one can use natural numbers to model the costs of constraints, the min operation to generate an ordering over them, and the sum operation to combine them. We thus have a constraint on each pair of variables stating that when the two colours are equal the cost is ∞, if just one of them is red the cost is 1, otherwise it is 2.

This extension has led to a high-level and flexible declarative programming formalism, called *Soft CLP* (SCLP), allowing to easily model and solve real-life problems. Roughly speaking, SCLP programs are logic programs where logical constants and operations are replaced by those of the

semiring. Consequently, assignments of variables to the items of the Herbrand universe yield the levels of satisfiability or the costs of the constraints.

We have applied the SCLP framework [MMH12] to the e-Mobility travel optimization problem described in [HZWS12], by modeling in Ciao [BCC$^+$97][8] two scenarios: the (i) trip; and (ii) journey optimization problems. A solution to (i) finds the best trip in terms of travel time and energy consumption, while (ii) determines the optimal sequence of trips, guaranteeing that the user reaches each appointment in time and that the state of charge of the electric vehicle never falls below a given threshold.

Besides optimizing trips and journeys of single users, that we can call local problems, the e-Mobility case study aims at solving global problems, involving large ensembles of vehicles. For such large problems, the solution is often unfeasible, with both SCLP and more efficient tools. To tackle these, we propose a coordination of declarative and procedural knowledge: the global problem is decomposed into several local problems, which can be separately solved by the SCLP implementation (e.g. [BCC$^+$97]), and whose parameters can be iteratively determined by a programmable coordination strategy. The latter guarantees a suboptimal, yet acceptable global solution.

Let us consider for example the parking optimization problem, which consists in finding the best parking lot for each vehicle of an ensemble in terms of three factors: the distance from the current location of the vehicle to the parking lot, the distance from the parking lot to the appointment location and the cost of the parking lot. Solving a global optimization procedure which assigns the best parking lot to each vehicle of the ensemble would be rather expensive, and also not flexible (replanning could require lots of time). So we propose to use SCLP to solve the local problems and some procedural language to program the orchestrator. In this setting, SCLP is convenient since the orchestrator will be able to access much more easily the parameters of its fact/clause-based declarative implementation than an ordinary imperative module structure.

In particular, the orchestrator could be programmed using an extension of SCEL or simply Java. The orchestrator, after receiving the requests from the vehicles which want to park, asks the SCLP tool to solve the local optimization problems, determining the best parking lot for each vehicle. Then, it verifies if the local solutions all together form an admissible global solution, that is, if each parking lot is able to satisfy the requests of the vehicles planning to park in it. If it is so, the problem is solved, otherwise the orchestrator queries the declarative knowledge again, but now by increasing the costs of the parking lots which received too many requests. The procedure is repeated, with suitable variations, until a global solution is found. Notice that in this way the orchestrator has a hypothetical, transactional behaviour, with the options of committing (a solution is found) or partially backtracking (on the parkings which are overfull).

Figure 8 shows the most important clauses of the CIAO program modeling the local problem of each vehicle. The $bestParkingLot$ clause determines the best parking lot for the vehicle by using the min predicate, which models the operation of the chosen semiring allowing us to choose the parking lot with the minimum cost. This clause also uses the $parkingLot$ predicate to obtain the cost of each parking lot in terms of the three factors described above. The head of the $parkingLot$ clause has indeed the following shape $parkingLot(CurrLoc, AppLoc, PL, C)$, where $CurrLoc$ represents the current location of the vehicle, $AppLoc$ is the location of the appointment, PL represents the parking lot and C the cost of the solution. This cost is given by a linear combination of the distance $DistCurrLocP$ from the current location to the parking lot, the distance $DistPA$ from the parking lot to the appointment location and the cost of the parking lot.

The Ciao program in Figure 9 instead models the parking lots. Also in this case we show the most important clauses. In particular, we have some facts modeling the costs of the parking lots and the $changeCost$ predicate allowing to increase the costs of a list of parking lots. It is used by the

[8] We would like to thank the Clip group for its technical support.

```
:-module(localOptimization,_,_).
:-use_module(parkingLot).
...
parkingLot(CurrLoc,AppLoc,PL,C):-
    parkingLot(PL, SN, PLLoc),
    distance(CurrLoc,PLLoc,DistCurrLocP),
    distance(PLLoc, AppLoc, DistPA),
    parkingCost(PL, PC),
    C .=. (X * DistPA) + (Y * PC)  + (Z * DistCurrLocP).

bestParkingLot(CurrLoc, AppLoc, BestPL):-
    findall([PL,Cpl], parkingLot(CurrLoc, AppLoc, PL, Cpl), ResL),
    min(ResL, BestPL).
...
```

Figure 8: The CIAO program modeling the local parking optimization problem.

```
:-module(parkingLot,_,_).
:-use_package(dynamic_clauses).
:- dynamic parkingCost/2.
...

parkingCost(p1, 1).
parkingCost(p2, 2).
...

changeCost([PL|PLL]):-
   retract_fact(parkingCost(PL, CPL)),
   C .=. CPL + 1,
   asserta(parkingCost(PL, C)),
   changeCost(PLL).

changeCost([]).
```

Figure 9: The CIAO program modeling parking lots.

orchestrator to change the costs of the parking lots which received too many requests.

3.2.7 Implementation and Deployment

Next steps in the EDLC, following the architectural design and detailed specification of component activities, is implementation and deployment. For these steps, we employ our DEECo (*Dependable Emergent Ensembles of Components*) component model [BGH+13b] to provide us with the relevant software engineering abstractions that ease the programmers' tasks.

A component in DEECo, features execution model based on the MAPE-K autonomic loop [KC03]. In compliance with SCEL, it consists of (i) well-defined knowledge, being a set of knowledge items and (ii) processes that are executed periodically in a soft real-time manner. The component concept is complemented by the first-class ensemble concept. An ensemble stands as the only communication mechanism between DEECo components. It specifies a *membership* condition, according to which components are evaluated for participation. The evaluation is based on the components' knowledge (their *attributes* in SCEL). An ensemble also specifies what is to be communicated between the participants, that is, the appropriate knowledge exchange function. Similar to component processes, ensembles are invoked periodically in a soft real-time manner. (See Figure 10 for an excerpts of components

```
component Vehicle features AvailabilityAggregator:
    knowledge:
        batteryLevel = 90%,
        position = GPS(...),
        calendar = [ POI(WORKPLACE, 9AM−1PM), POI(MALL, 2PM−3PM), ... ],
        availabilities = [ ],
        plan = {
            route = ROUTE(...),
            isFeasible = TRUE
        }
    process computePlan:
            in plan.isFeasible, in availabilities,
            in calendar, inout plan.route
        function:
            if (!plan.isFeasible)
                plan.route ← Planner.computePlan(calendar, availabilities)
        scheduling: periodic( 5000ms )
    ...
...
ensemble UpdateAvailabilityInformation:
    coordinator: AvailabilityAggregator
    member: AvailabilityAwareParkingLot
    membership:
        ∃ poi ∈ coordinator.calendar:
‘   distance(member.position, poi.position) ≤ TRESHOLD &&
            isAvailable(poi, member.availability)
    knowledge exchange:
        coordinator.availabilities ← { (m.id, m.availability) | m ∈ members }
    scheduling: periodic( 2500ms )
...
```

Figure 10: Examples of identified DEECo components & ensembles.

and ensembles descriptions as found in the e-Mobility case study.)

In order to bring the above abstractions to practical use we have used jDEECo[9] – our reification of DEECo component model in Java. In jDEECo, components are intuitively represented as annotated Java classes, where component knowledge is mapped to class fields and processes to class methods. Similarly, appropriately annotated classes represent DEECo ensembles.

Once the necessary components and ensembles are coded, they are deployed in jDEECo runtime framework, which takes care of process and ensemble scheduling, as well as low-level distributed knowledge manipulation.

3.2.8 Evaluation

Having described the application of EDLC to the e-Mobility case study, we relate it in this section to other approaches having the same aim and we describe benefits that we have observed in performing the case study. In particular, we structure this section along three main topics addressed in the case study, namely (i) requirements engineering and architectural design, (ii) modeling of activities, (iii) programming and deployment.

As to requirements engineering and architectural design in the area of autonomic systems, the most recognized approaches are KAOS [Lam08] and Tropos [BPG+04]. Similar to our approach, they fall into the category of goal modeling and elaboration, especially in the area of agent-based systems. In our experience, they provide a very solid ground for requirements engineering, but fall short to an extent when continuous control with self-adaptivity (as in the case of e-Mobility case study) is sought for. For this reason, we have employed SOTA and IRM, which are centered around the notion of

[9] http://github.com/d3scomp/JDEECo

continuously "striving to achieve" and thus make the reasoning about a guided evolution of a system easier.

As for the activity modeling, our approach builds on the body of work carried out in coordination languages (e.g., KLAIM [DNFP98]) and process algebras. However, it extends it by providing a tailored semantics to describe and reason about cooperating groups of components (i.e. ensembles). In the same vein, SCLP builds on the experience with constraint solving, but adds the option of soft-constraints and integration with SCEL. Indeed, in the e-Mobility case study, we found the interplay of SCEL with SCLP very useful for description of mutually related activities of interaction and coordination among vehicles combined with finding a trade-off between local-global optimums (reflecting the need of harmonizing the selfish and cooperative concerns of vehicles in the case-study).

Finally, at the programming and deployment stage, our approach has been backed up by DEECo component model and its Java-based reification jDEECo. In this respect, it is possible to find a plethora of component models and SOA-based approaches (e.g. SCA, Fractal, OSGi). However, these typically fall short in well-defined dynamicity (as captured by the ensembles) as well as in autonomicity and self-adaptation capabilities (as featured by the special design of components as distributed MAPE-K based entities). Similar problems apply even to the agent-based approaches with their Belief-Desire-Intention model (e.g., JADE). On the other hand, the explicit support of DEECo for ensembles and components – based on knowledge and cyclic activities – proved to make the transition from SOTA/IRM-based design (together with activities captured by SCEL/SCLP) to runtime very smooth.

3.3 Swarm Robotics

Swarm robotics calls for advanced techniques to provide adaptive, autonomous, self-aware and intelligent behavior. One software solution to such requests is an ensemble-based approach that structures a complex control system into dynamic ensembles of relatively simple system elements, called *service components*. The dynamism and autonomous nature of the system elements is modeled by the communication/distribution principle. This principle can be realized through knowledge- and predicate-based mechanisms, which allow for run-time evaluation of communication and connection rules among the system elements.

The work in robotics case study revolves around a family of scenarios that pose interesting design challenges. The basic idea is that a disaster happened in a building. Part of the building has collapsed, trapping a number of victims inside. To make matters worse, a radiation source is present, which damages robots as well as humans. To prevent harm to further human lives, a team of robots is deployed in a specific area—the *deployment area*. From that location, the robots must explore, search for victims, and coordinate to save the victims as quickly as possible, while avoiding damage from radiation.

During the third year of the project, we discussed and identified a number of interesting variants for this basic template. Their detailed description is reported in D7.3. In brief, we distinguish between *easy* and *hard exploration*, depending on the structure of the environment. In the first case, robots only move within a corridor. In the second one, the environment is more complex and the robots are required to construct a spatial representation of the environment, either in the form of an individual map, or in the form of a network of landmark robots. In addition, the robots must construct a wall to protect themselves and the victims from radiation.

In the following discussion, we describe the EDLC phases used for requirement analyses (Sec. 3.3.1), high-level modeling (Sec. 3.3.2), SCEL modeling, (Sec. 3.3.3), awareness mechanisms (Sec. 3.3.4), and deployment (Sec. 3.3.5), focusing on the tools that realize each phase—SOTA, MESSI, SCEL, Iliad, and jRESP/ARGoS.

3.3.1 Requirement Analyses

The requirement analysis phase is the first step towards a sound implementation of a solution for the robotics scenario. The input of this phase is a high-level, often non-formal description of the scenario, such as that reported in D73. The output of this phase is a semi-formal description of the system dynamics, as well as a set of adaptation patterns that fit the system requirements.

In the case under study, the robots behave as a coordinated swarm. For this reason, the patterns that take inspiration from swarm robotics, such as the *Reactive Stigmergy Service Components Ensemble Pattern* or the *Cognitive Stigmergy Service Components Ensemble* [Puv12], are the most natural to describe the system. These patterns, as it can be found in their **Context**, describe systems composed of *a large amount of components acting together*, in which *the components are simple* and *the environment is frequently changing*. A number of studies confirm that through these patterns the system can achieve high performance (e.g. [PPC$^+$13], [PCZ13]).

An interesting aspect of this scenario is that a victim may be too heavy to be transported to safety by a single robot. Thus, cooperation among multiple robots is required. To achieve coordination, multiple alternatives are possible, spanning from leader-based to completely decentralized approaches. We maintain that a SOTA specification helps selecting the right approach.

For example, in the *Reactive Stigmergy Service Components Ensemble Pattern*, the goals **G** and utilities **U** are treated disjointly from a component to another—each *service component* has its own goals and utilities:

- *Goals*: $G_{SC_1}, G_{SC_2}, \ldots, G_{SC_n}$
- *Utilities*: $U_{SC_1} = U_{SC_2} = \ldots = U_{SC_n}$

while, in a pattern whereby a direct coordination is needed to manage adaptation, they can be expressed as:

- *Goals*: $G_{SC_1} \bigcup G_{SC_2} \bigcup \ldots \bigcup G_{SC_n}$
- *Utilities*: $U_{SC_1} \bigcup U_{SC_2} \bigcup \ldots \bigcup U_{SC_n}$

The SOTA description offers the means to select the pattern to use for a specific task because, describing functional and non-functional requirements, it allows the developer to identify which pattern is more similar to the target system. Moreover, self-expression [PPC$^+$13] can be used to allow a system to satisfy tasks for which it was not created: recognizing the new functional and non-functional requirements, a new pattern can be selected, and, using self-expression mechanisms (as described in [Puv13]), the dynamic change of the pattern can be done. For example, one could envision that, when the robots must cooperate to transport a victim, or to construct a map of the environment, they could switch to a pattern like the *Centralized AM Service Components Ensemble Pattern*, or the *P2P Negotiation Service Components Ensemble Pattern* to perform the task, and then switch back to their original behavioral pattern. To this aim, a self-expression mechanism that can be used is the *role-based approach* [PNA$^+$13, Puv13] whereby a robot dynamically changes its role from "environment explorer" to, e.g., "group leader" or "follower". The interesting aspect of this role-based approach is that, by simply changing roles, robots can also form coordinated groups with clear leader-follower assignments.

To understand how SOTA and adaptation patterns work in the robotic scenario, we must describe the scenario in SOTA. The SOTA space **S** includes:

- Spatial aspects related to, e.g., landmark placement and victim locations;

- Robot-specific internal aspects, such as battery consumption;
- Ensemble-related aspects, such as the distribution of the behaviors of the robots over time.

As a robot performs its actions, its position in the SOTA space (i.e., its *SOTA state*) changes accordingly.

In SOTA, a goal is expressed in terms of a specific target SOTA state. For instance, when a robot reaches a victim, the former must have enough battery life in order to safely transport the latter. This can be expressed as a SOTA state in which the spatial sub-state of the robot is within a suitable range to the victim, the battery level sub-state is above a suitable threshold, and the behavior sub-state corresponds to the "transport" behavior. In SOTA, this can be described as follows:

$$
\begin{aligned}
Grobot_i = \quad \{ \quad & G^{pre} = local\_position; \\
& G^{post} = new\_location \bigcup victim\_rescue(carry, mark\_position); \\
& U = battery\_level > min\_level \\
\} &
\end{aligned}
$$

This system can be cast as an instance of the *P2P Negotiation SCE Pattern*, which revolves around an implicit autonomic feedback loop for each proactive component. In this pattern, the components need to directly communicate to propagate adaptation mechanisms and to share knowledge. The main advantage of the *P2P Negotiation SCE Pattern* is that no centralised control is present, thus excluding the single-point-of-failure issue. Each robot is aware of its battery state and has its private goal. Moreover, because a robot does not know where the victim is and does not have the possibility of mapping all the environment, each robot is able to propagate messages and exchange information with neighbours, to better adapt its behaviour and find the victim. Because the most important utility of a robot is to maintain a high battery level, each robot has the sub-goal of exploring only a portion of the space while looking for victims. The *P2P Negotiation SCE Pattern* allows a robot that reaches an unexplored position to send a message to its neighbours and communicate the state of this location (victim found or new point of connection), and to ask for a new robot to complete the task.

Another candidate pattern to design this system could have been the *Reactive Stigmergy SCE Pattern* (described before). However, this pattern does not support explicit coordination among robots, so, using this pattern, robots can sense information propagated in the environment, but cannot share a common adaptation mechanism in order to find and rescue a victim.

3.3.2 Modeling and validation with MESSI

The output of the Requirement Engineering phase is a SOTA specification together with one or more adaptation patterns identified as suitable for the robotic scenario to be tackled. The first step of the Modeling phase consists of producing an abstract specification of a strategy for the robot swarm, suitable to achieve the SOTA goal and satisfying the corresponding utilities. The specification is presented in a form that is quite a standard for describing distributed strategies for swarm robotics (see [OGCD10]), namely as a diagram representing a finite state machine (like the one in Figure 11). One or more such diagrams can be produced, specifying different strategies or variations thereof.

These diagrams constitute the input of MESSI [MES] (*"Maude Ensemble Strategies Simulator and Inquirer"*), a tool for early prototyping and analysis of robotic strategies. Within the EDLC, MESSI is placed conceptually in the Validation phase of the Design Cycle, and its key features are presented in JD3.1 and D6.3. MESSI allows one to build a high-level but executable model of the specified strategy, and to validate it via animated simulation and quantitative analysis. The outcome of this prototyping and validation step may serve as a feedback to the Requirement Engineering phase, should some ambiguity or inconsistency in the original requirements be detected. Otherwise, the comparison of

different strategies will produce a streamlined finite state machine, that might be mapped into suitable SCEL terms, and thus passed to the jRESP simulation and analysis environment. An early analysis on prototypes, even if performed on an abstract representation of the real system, can at least speed-up testing and debugging, and dispense the programmers from coding lowest-performance strategies.

MESSI conforms to the CODA conceptual framework (for *Control Data*, the key feature of the methodology), a white-box approach to the specification of self-adaptive systems, aiming at providing simple yet precise guidelines for their clean structuring. We proposed it in a few papers [BCG$^+$12a, BCG$^+$a] and presented it already in deliverables D2.2 and JD2.1. Briefly, the main idea is that the programmer is in charge of separating the adaptation and the application logic, by explicitly identifying the control data, i.e. the data of the system whose modification triggers an adaptation. MESSI is implemented with the reflective language Maude, and it supports a hierarchical software architecture style called "reflective russian dolls", which we have found well-suited for modeling autonomic systems with self-* features [BCG$^+$12b, BCG$^+$b]. We decided to adopt an implementation using available, state-of-the-art tools, to be able to test immediately our proposal as a component of the life cycle. We foresee its reengineering, by using the chain of tools and techniques now developed by ASCENS, the MISSCEL simulator reported in JD3.1 and D6.3, as well as in D1.3, being a case at hand.

As a concrete example to test the effectiveness of our approach, we addressed an "*easy exploration*" variant of the robotic scenario where no construction is needed. We further assume that the victim is either too heavy for a single robot or that it has to be protected. More precisely, we currently assume that a victim robot has to be surrounded by eight robots in order to be rescued. This number might be parametrized, depending on the weight of the victim and/or on the strength of the rescue-robots. So the problem is to identify a victim, attract other robots to completely surround it aggregating in a circle, and. when the assembly is ready, move all together towards a light, representing a safe location. Noteworthy, the strategy of Figure 11 is part of the outcome of the work done by a group of three students during the *International AWARENESS summer school on Self-Awareness in Autonomic Computing Systems (AWASS 2012)* [10]. It is worth noticing that none of the students were expert in robotics, self-assembly behaviours or swarm computing. This shows the ease of use of our framework.

The strategy depicted in Figure 11 is executed concurrently and independently by all rescue-robots in the swarm. Each state is implemented by a basic MESSI controller which is a module (a set of rules) realizing a predefined behaviour: for example, MOVE_IN_ANY_DIRECTION implements random walking; IDLE lets the robot stay idle; and robots in state GRAB_ADMISSIBLE_LED try to grab a robot with red LEDs. Transitions among states represent changes of behaviour, triggered by the condition labeling the arrow. They are implemented using reflection and are seen as adaptations of the robot, because they correspond to a change of control data, i.e., of the MESSI controller currently under eecution. Briefly, a robot starts in the AGG state and, walking around, it changes to SA when it sees a victim. When it grabs the victim (states WT or GWT) it changes its LEDs to magenta and stops. When another robot sees a magenta LED, it changes state to SBB, where the OUTLANK_EFFECT controller lets it explore the surroudings looking for the victim. States GWT, WT, PT1 and PT2 implement a token passing protocol to check that the victim is completely surrounded by robots. When this happens, all robots are in state SH and, for the sake of conciseness, we assume that after a timeout they change state to CP where they move together towards a light source.

The comparison of the performances of different strategies implemented in MESSI can be carried out using a parallel statistical model checker. We do not have results about the above collective rescue scenario yet, as we currently focused on obstacle avoidance (e.g. hole-crossing while navigating towards a light source) and morphogenesis (where robots assemble to form predefined shapes) scenarios. Indeed, the procedure proved adequate in identifying some weaknesses in the specification

[10] `http://www.aware-project.eu/awareness-training/awass-2012/`

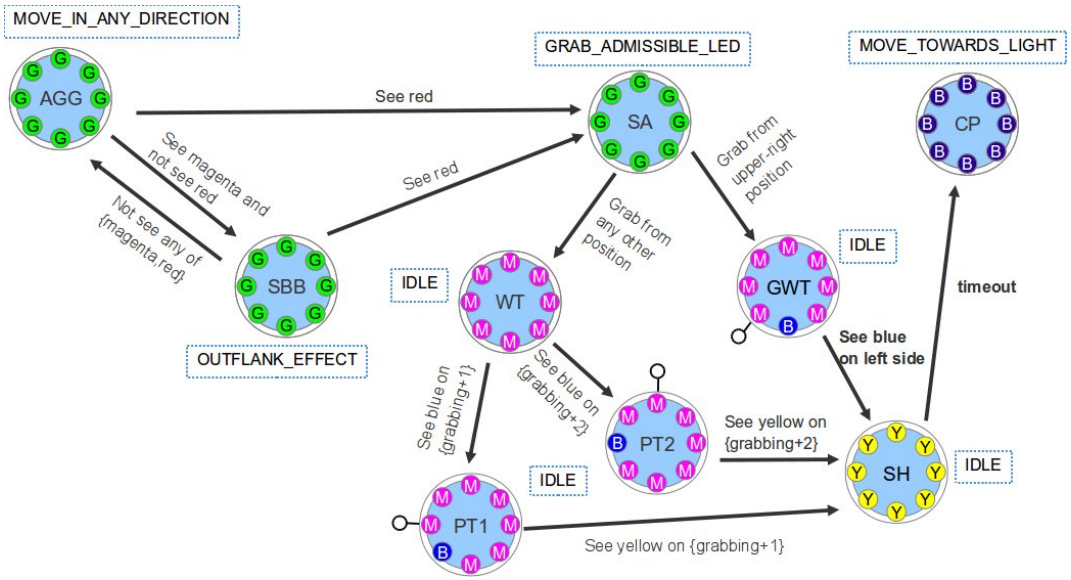


Figure 11: A self-assembly strategy for collective rescue scenarios.

of the *basic self-assembly response strategy* proposed in [OGCD10], as reported in [BCG+b]. Successive rounds of specification tuning and analysis provided a valuable feedback, and allowed for the development of sound and sometimes more efficient strategies for such scenarios.

3.3.3 SCEL Modeling and jRESP Programming

In this section, we outline how SCEL [DLPT13, DFLP11b] can be used to model the robotic scenarios. In the considered model, each robot is described via a SCEL component.

To model an autonomic system in SCEL, the first step is to identify the basic attributes that characterize the relevant aspects of Autonomic Components (ACs) involved. ACs are entities with dedicated knowledge units and resources. ACs can cooperate while playing different roles. Awareness is enabled by providing ACs with information about their own state and behavior; such information is stored in each AC's knowledge repository. These repositories also allow ACs to store and retrieve information about their working environment. Ultimately, this information is used to redirect and adapt their behavior. Each AC is equipped with an *interface*, consisting of a collection of *attributes*, such as provided functionalities, spatial coordinates, group memberships, trust level, response time, etc. Attributes are used by the ACs to dynamically organize themselves into *autonomic-component ensembles* (ACEs). These are sets of ACs featuring goal-oriented execution. ACEs are highly dynamic and flexible, and not curbed by rigid structures. These attributes are typically identified during the requirements analysis and are related to the set of *goals* and *utilities* that are part of SOTA specification.

In the robot scenarios, one of the main attributes is the task that a robot is performing. For instance, in the *hard exploration* variant, when individual mapping is not possible, possible values for this attribute, that we call task, are: *exploring*, *networking*, *searching* and *rescuing*.

The value of attribute task is determined by what the robot sensors perceives from the enclosing environment. The schema of robot behaviour is the following: a group of robots starts in task exploring and performs a random walk across the area. A robot switches to task networking when either it perceives a *victim*, or it detects a given number of robot invoked in task networking. The following

code segment is the SCEL process identifying the exploring task:

$$
\begin{array}{lll}
P_{exp}(id) & \triangleq & //\text{While executing this behaviour a robot moves following a random walk.} \\
 & & \mathbf{get}(\text{“}victim\_sensed\text{”})@\mathsf{self}. \\
 & & \quad //\text{When a victim is perceived, the robot stops moving} \\
 & & \quad \mathbf{put}(\text{“}stop\text{”})@\mathsf{self}. \\
 & & \quad //\text{The fact that a victim has been directly found is published in the local knowledge} \\
 & & \quad \mathbf{put}(\text{“}victim\text{”}, 0, id)@\mathsf{self}. \\
 & & \quad \mathbf{nil} \\
 & & + \mathbf{qry}(\text{“}victim\text{”}, ?d, ?r)@true. \\
 & & \quad //\text{A robot also stops the presence of a victim is inferred from another robot} \\
 & & \quad \mathbf{put}(\text{“}stop\text{”})@\mathsf{self}. \\
 & & \quad //\text{In this case the the robot publish in the knowledge} \\
 & & \quad //\text{that a victim has been indirectly perceived} \\
 & & \quad \mathbf{put}(\text{“}victim\text{”}, d+1, id)@\mathsf{self}. \\
 & & \quad \mathbf{nil}
\end{array}
$$

During the exploration phase, robots publish information about the *distance* (in terms of number of *hops*) to the victims in the local knowledge. This information will be used by the other robots (all in state *searching*). A victim can be discovered by following the chain of decreasing *hops*. When a robot at distance 0 from a victim is found, a robot starts the rescue phase. The processes implementing the *searching* procedure are reported below:

$$
\begin{array}{lll}
P_{search}() & \triangleq & //\text{Looks for a robot that directly or indirectly perceives a victim} \\
 & & \mathbf{qry}(\text{“}victim\text{”}, ?d, ?r)@true. \\
 & & P_{follow}(r, d)
\end{array}
$$

$$
\begin{array}{lll}
P_{follow}(r, d) & \triangleq & \mathbf{if}\ (d == 0)\{ \\
 & & \quad //\text{A robot that directly perceives a victim has been found.} \\
 & & \quad P_{resc}() \\
 & & \} \\
 & & //\text{Move towards robot } r \\
 & & \mathbf{put}(\text{“}towards\text{”}, r)@\mathsf{self}. \\
 & & //\text{Search the next robot in the path to the victim} \\
 & & \mathbf{qry}(\text{“}victim\text{”}, d-1, ?r)@true. \\
 & & P_{follow}(d-1, r)
\end{array}
$$

Starting from a SCEL specification we can easily obtain the corresponding jRESP code that provides a Java implementation of the robot controller. The implementation of jRESP fully relies on the SCEL's formal semantics. This close correspondence enhances confidence on the behavior of the jRESP implementation of SCEL programs, once the latter have been analyzed through formal methods made possible by the formal operational semantics. Unfortunately, jRESP programs cannot be executed on a real robot. However, we can simulate the behavior of the system by relaying on the simulation environment provided by jRESP.

In particular, to support analysis of adaptive systems specified in SCEL, jRESP provides a set of classes that permits simulating jRESP *programs*. These classes allow the execution of *virtual components* over a simulation environment that is able to control component interactions and to collect relevant simulation data.

By relying on jRESP simulation environment, a prototype framework for *statistical model-checking* has been also developed. Following this approach, a randomized algorithm is used to verify whether

the implementation of a system satisfies a specific property with a certain degree of confidence. Indeed, the *statistical model-checker* is parameterized with respect to a given *tolerance* ε and *error probability* p. The used algorithm guarantees that the difference between the value computed by the algorithm and the exact one is greater than ε with a probability that is less than p.

The model-checker included in jRESP can be used to verify *reachability properties*. These permits evaluating the probability to reach, within a given deadline, a configuration where a given predicate on collected data is satisfied.

3.3.4 Awareness Mechanisms

As described in Sect. 3.3.1, the requirement analysis phase of the EDLC results in a state-space model of the system's behavior, and in goals and utilities that describe the desired behavior of the system. To convert the abstract requirements into an awareness-based solution that can be executed on the *Iliad* runtime, the main tasks are:

1. State and action abstraction, i.e., reducing the abstract state-space to a manageable size, in such a manner that the reasoners of the awareness mechanism can still decide whether the ensemble's goals have been reached, and which effects various actions have on the state;

2. Developing strategies (either internal to the awareness mechanism, or external) that can determine which actions must be taken in order to reach the goals determined in the analysis phase.

Often, the awareness mechanism will be used by SCEL/jRESP programs that serve as autonomic manager and managed elements, as described in the previous section. In this context, the SCEL programs are responsible for high-level behaviors, control, and synchronization. The distribution of work between awareness mechanism and SCEL programs can be handled in a flexible manner: a SCEL program might, e.g., only use the reinforcement-learning component of *Iliad* to perform state estimation, and perform all computations concerning the actions that should be performed on its own; a different solution might instead use the HTN planner of *Iliad* to compute the actions to execute, so that the SCEL program serves only as "action executor" that performs almost no computation. By using the run-time cycle of the EDLC, it is also possible to combine these two solutions: the system can execute (efficient) jRESP code in situations that were planned for at design time, or that it has already encountered and solved in the past. When an unexpected situation appears, or the measured performance degrades, the SCEL program acting as autonomic manager can ask the awareness mechanism to compute a new strategy (using potentially expensive reasoning mechanisms). After obtaining a solution, the autonomic manager can trigger an adaptation process that places the new strategy into the library of known solutions available to the managed elements, so that similar situations can be handled efficiently in the future.

In addition to these general considerations, the robotics scenario provides several challenges for the awareness mechanism (and any formalism that is based on high-level descriptions of the domain or on reasoning): The marXbots provide relatively limited memory and computational power, so that running complex reasoning mechanisms on the robots themselves is practically infeasible. The data delivered by most sensors is noisy, and because of the limited computational power available, many state estimation techniques that are used on larger robots cannot be performed on the marXbot. Thus, it is often not possible to obtain precise information about the robot's state, and decisions must be made using a probabilistic state estimate.

Even with these restrictions, an awareness mechanism based on the SOTA model can provide significant improvements to the system's performance. We have run several experiments to explore awareness mechanisms. In one of these experiments, a simulated robot executes a rescue operation in a slightly simplified version of the *hard exploration* scenario where no construction is needed and the

```
(define-constant all-directions '(N NE E SE S SW W NW))
(defun* nav (loc)
  "nav LOC
   Navigate to location LOC.  Repeatedly choose among the navigation
   actions until the robot reaches LOC."
  (until (equal (robot-loc) loc)
    (with-choice navigate-choice (dir (remove-obstacles all-directions))
      (action navigate-move dir))))
```

Figure 12: High-level partial program for navigation in the rescue scenario

environment has a maze-like topology with mostly narrow corridors: the robot navigates to a victim (whose position is known), picks up the victim, and then navigates back to a rescue zone. The robot is not provided with a map of the environment. To avoid the complexities of Simultaneous Localization and Mapping (SLAM), for the time being we assume that the robot can determine its position and orientation, e.g., by using a GPS receiver. The robot incurs a penalty for bumping into obstacles while navigating or for driving into areas with radiation, but it cannot reliably sense obstacles and radiation; instead, it can only determine the problem when it receives a penalty after moving. This is a realistic assumption for a robot relying on local sensors only. The robot can infer whether the penalty was due to an obstacle or radiation by checking whether its position is unchanged after receiving a penalty; however, the robot has a certain chance of slipping and moving in a different direction than intended, so this check is not completely reliable. While the experiment removes uncertainty about the robot's location and does not take into account moving obstacles, it addresses many issues of the full *hare exploration* scenario, and is an important first step towards developing an awareness mechanism that can drive the run-time cycle of the EDLC in robotic scenarios.

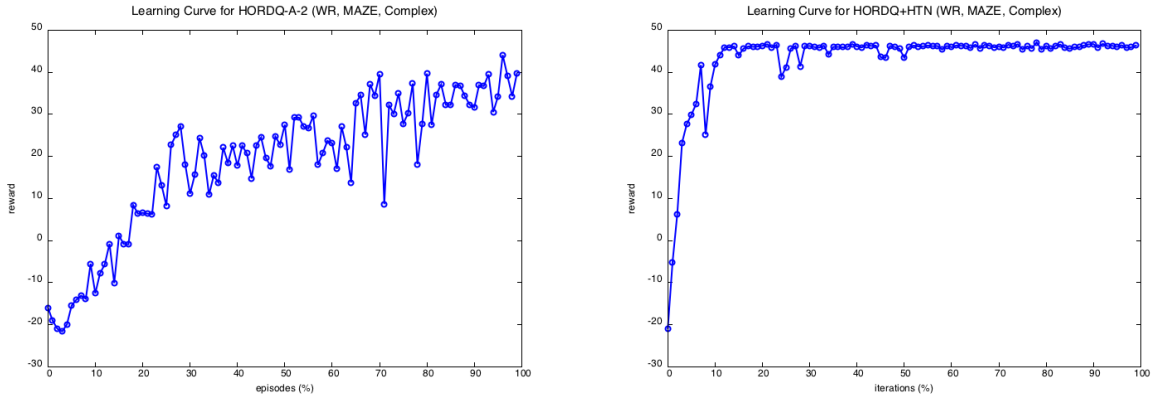
Fig. 12 shows the partial program (i.e., an internal strategy) used by a simple awareness mechanism based on reinforcement learning. This program repeatedly picks an action (move north, move north-west, etc.) until the robot has reached the desired target location; it is a partial program since the program itself does not specify *which* of the possible actions should be taken; this choice is performed by a reinforcement learning mechanism. Fig. 12 shows the action abstraction performed by the awareness mechanism: the robot's motion is represented by a compass point only. The label `navigate-choice` is responsible for connecting the partial program to the reinforcement learning mechanism which also performs state abstraction; the label `navigate-move` is used by the runtime to execute the desired action, e.g., by placing a tuple into the tuple space of the controlling SCEL program.

Figure 13a shows a relatively typical learning curve when using the program of Fig. 12. Even with this extremely simple partial program and the default reinforcement learning algorithm used by *Iliad*, the robot will reliably rescue the victim (have a score > 0) after ca. 3000 simulated episodes.[11] By taking into account knowledge about similar routes and performing a limited amount of HTN-planning, the performance of the awareness mechanism can be significantly increased while still leaving it lightweight enough to execute on individual marXbots (see Figure 13b).

3.3.5 Deployment

The final phase of the EDLC involves the deployment of the robot behaviors on the robots. This phase is the most critical because it is usually the most economically expensive, time-consuming, and risky.

[11] This low number of episodes relative to the state space is possible because of the maze-like structure of the environment. For this simple program, performance degrades if the environment contains many open areas.



(a) Simple learning

(b) Knowledge-based learning + planning

Figure 13: Learning curves for single-robot rescue operation in a simplified, maze-like *hard-exploration* environment with slippery floor (random exploration, approx. 15000 episodes, using discrete state abstraction with approx. 6400 states, max. possible avg. performance ≈ 47)

For this reason, deployment is usually performed in two distinct sub-phases. The first sub-phase consists in testing the behaviors in accurate physics-based simulations. These simulations must include as much detail as possible, so as to minimize catastrophic issues in the next sub-phase. The next deployment sub-phase consists in testing the behaviors on the real platforms.

Deployment is likely to uncover problems that have been overlooked in the previous phases of the EDLC. Thus, the outcome of the deployment activities can be exploited as feedback to refine the analysis of the system as of the previous EDLC phases, fostering new cycles of design. In ASCENS, the deployment phase is completed using either ARGoS framework [PTO+12][12] or jRESP[13].

ARGoS offers facilities both to simulate large-scale robots swarms as accurately as necessary, and to deploy the refined behaviors onto the real robots without modification. The robot behaviors can be written either in C++ or in Lua. For instance, the *explore* behavior of the robots for the landmark-based navigation approach described in D73 is reported in Fig. 14.

In addition, by using the Hexameter communications infrastructure[14], ARGoS can integrate awareness mechanisms developed using the Iliad runtime either directly on individual robots or remotely over a network connection.

Alternatively, jRESP is a Java runtime environment providing a framework for developing autonomic and adaptive systems according to the SCEL paradigm. Specifically, jRESP provides an API that permits using in Java programs the SCEL's linguistic constructs for controlling the computation and interaction of autonomic components, and for defining the architecture of systems and ensembles. jRESP provides specific components that can be used to simulate SCEL programs.

[12] `http://iridia.ulb.ac.be/argos`

[13] jRESP (Java Run-time Environment for SCEL Programs) website: `http://jresp.sourceforge.net/`.

[14] `https://github.com/thomasgabor/hexameter/` and `https://github.com/hoelzl/Hexameter`

```
--[[ Logic of state EXPLORE ]]
function rescuer:explore()
 -- State transition logic
 -- Idea: Become a landmark if you get too far from the closest known landmark
 -- Be sure to be out of the nest (landmarks are useless inside the nest)
 if rescuer:is_out_of_nest() then
  -- Get the landmarks around
  local landmarks = rescuer:landmarks_in_range()
  if landmarks then
   -- Get the data of the closest landmark
   local dist = RAB_RANGE
   local marker
   local is_victim_landmark = false
   for i = 1, #landmarks do
    if landmarks[i].range < dist then
     dist = landmarks[i].range
     marker = landmarks[i].data[3]
     is_victim_landmark = (landmarks[i].data[2] == RESCUER_STATE__VICTIM_LANDMARK)
    end
   end
   -- Are we getting too far from the closest?
   if (not is_victim_landmark) and
      (dist > 0.8 * RAB_RANGE) then
    -- The closest landmark is getting too far
    -- Become a landmark!
    rescuer:become_landmark(marker)
    return
   end
  end
 else
  -- Explorer got back to the nest
  -- Switch back to exiting state
  rescuer:switch_to_exiting()
  return
 end
 -- State logic
 -- Wander in the environment
 local repulsion = rescuer:repulsion_vector()
 if(repulsion.x*repulsion.x+repulsion.y*repulsion.y > 0.001) then
  rescuer:vector_to_wheel_velocity_noscale(repulsion)
 else
  robot.wheels.set_velocity(5,5)
 end
end
```

Figure 14: The *Explore* behavior for the landmark-based approach in D73, written in Lua on ARGoS.

4 Conclusions

In this deliverable we presented a software development life cycle for autonomic systems. Its aim is to support developers dealing with self-awareness and self-adaptation in ensembles, talking into account environmental situation. A distinguishing feature of the double-wheeled life cycle is the feedback loop from runtime to design (in addition to the feedback loops at runtime provided by classical approaches for self-adaptive engineering). It is also important to remark that our life cycle relies on foundational methods used for the verification of the expected behaviour; indeed this provides this way another feedback loop that allows for improvement of the software. We illustrated how the life cycle can be instantiated using a set of languages, methods and tools developed within the ASCENS project.

The proof of concept of the life cycle was performed for the three domains of the ASCENS case studies: robot swarms, cloud computing and e-mobility. Although the selected scenarios of these case studies are quite different, from the software engineering point of view similar methods and techniques can be applied to ensure awareness and self-adaptation. The main commonalities regarding techniques and tools are:

- Requirements engineering based on SOTA (State-of-the-Affairs)
- Modeling and programming of the scenarios with SCEL (Software Component Ensemble Language)
- Implementation supported by the jRESP (Java runtime environment) and DEECo (Dependable Emergent Ensembles of Components)
- Use of adaptation patterns
- POEM awareness mechanisms

Within the scope of ASCENS there have been developed many other methods, techniques and tools also applicable in the EDLC but which were not mentioned in this deliverable. The reader is referred to deliverables of work packages WP1 to WP8 and in particular to the ASCENS Joint Deliverable on Verification Results Applied to the Case Studies (JD3.1) [Be13] for further scenarios as well as methods, tools and techniques used in the validation and verification phase of the EDLC.

References

[ALLS13] Michele Amoretti, Alberto Lluch-Lafuente, and Stefano Sebastio. A cooperative approach for distributed task execution in autonomic clouds. In *PDP*, pages 274–281. IEEE Computer Society, 2013.

[AZ12] D. B. Abeywickrama and F. Zambonelli. Model Checking Goal-Oriented Requirements for Self-Adaptive Systems. In *Proc. of ECBS*, pages 33–42. IEEE, April 2012.

[BBD+12] Saddek Bensalem, Marius Bozga, Benoît Delahaye, Cyrille Jégourel, Axel Legay, and Ayoub Nouri. Statistical Model Checking QoS Properties of Systems with SBIP. In Tiziana Margaria and Bernhard Steffen, editors, *ISoLA (1)*, volume 7609 of *LNCS*, pages 327–341. Springer, 2012.

[BBH+12] Lubomir Bulej, Tomas Bures, Vojtech Horky, Jaroslav Keznikl, and Petr Tuma. Performance Awareness in Component Systems: Vision Paper. In *Proceedings of COMPSAC 2012*, COMPSAC '12, 2012.

[BBHK13] Lubomír Bulej, Tomáš Burea, Vojtčch Horký, and Jaroslav Keznikl. Adaptive deployment in ad-hoc systems using emergent component ensembles: vision paper. In *Proceedings of the 4th ACM/SPEC International Conference on Performance Engineering*, ICPE '13, pages 343–346, New York, NY, USA, 2013. ACM.

[BBK+12] Lubomir Bulej, Tomas Bures, Jaroslav Keznikl, Alena Koubkova, Andrej Podzimek, and Petr Tuma. Capturing Performance Assumptions using Stochastic Performance Logic. In *Proc. 3rd Intl. Conf. on Performance Engineering*, ICPE'12, Boston, MA, USA, 2012.

[BBNS09] Saddek Bensalem, Marius Bozga, Thanh-Hung Nguyen, and Joseph Sifakis. D-Finder: A Tool for Compositional Deadlock Detection and Verification. In Ahmed Bouajjani and Oded Maler, editors, *CAV*, volume 5643 of *LNCS*, pages 614–619. Springer, 2009.

[BBS06] Ananda Basu, Marius Bozga, and Joseph Sifakis. Modeling Heterogeneous Real-time Components in BIP. In *SEFM*, pages 3–12. IEEE Computer Society, 2006.

[BCC+97] F. Bueno, D. Cabeza, M. Carro, M. Hermenegildo, P. López-García, and G. Puebla. The Ciao Prolog System. Reference manual. Technical Report CLIP3/97.1, School of Computer Science, Technical University of Madrid (UPM), 1997.

[BCG+a] Roberto Bruni, Andrea Corradini, Fabio Gadducci, Alberto Lluch-Lafuente, and Andrea Vandin. A conceptual framework for adaptation. *ACM Transactions on Autonomous and Adaptive Systems*. submitted.

[BCG+b] Roberto Bruni, Andrea Corradini, Fabio Gadducci, Alberto Lluch-Lafuente, and Andrea Vandin. Modelling and analyzing adaptive self-assembly strategies with maude. *Science of Computer Programming*. to appear.

[BCG+12a] Roberto Bruni, Andrea Corradini, Fabio Gadducci, Alberto Lluch-Lafuente, and Andrea Vandin. A Conceptual Framework for Adaptation. In Juan de Lara and Andrea Zisman, editors, *FASE*, volume 7212 of *LNCS*, pages 240–254. Springer, 2012.

[BCG+12b] Roberto Bruni, Andrea Corradini, Fabio Gadducci, Alberto Lluch-Lafuente, and Andrea Vandin. Modelling and analyzing adaptive self-assembly strategies with maude. In Franciso Durán, editor, *WRLA*, volume 7571 of *Lecture Notes in Computer Science*, pages 118–138. Springer, 2012.

[BdSIY13] Saddek Bensalem, Lavindra de Silva, Félix Ingrand, and Rongjie Yan. A verifiable and correct-by-construction controller for robot functional levels. *CoRR*, abs/1309.0442, 2013.

[Be13] Saddek Bensalem and Jacques Combaz (eds.). Verification Results Applied to the Case Studies, November 2013. ASCENS Join Deliverable JD3.1.

[BGH+13a] Tomas Bures, Ilias Gerostathopoulos, Petr Hnetynka, Jaroslav Keznikl, Michal Kit, and Frantisek Plasil. DEECO: An Ensemble-Based Component System. In *Proceedings of the 16th International ACM Sigsoft symposium on Component-based software engineering*, CBSE '13, pages 81–90, New York, NY, USA, 2013. ACM.

[BGH+13b] Tomas Bures, Ilias Gerostathopoulos, Petr Hnetynka, Jaroslav Keznikl, Michal Kit, and Frantisek Plasil. DEECo an Ensemble-Based Component System. In *Proc. of CBSE '13*, Vancouver, Canada, 2013. ACM.

[BGL+11] Saddek Bensalem, Andreas Griesmayer, Axel Legay, Thanh-Hung Nguyen, and Doron Peled. Efficient Deadlock Detection for Concurrent Systems. In Satnam Singh, Barbara Jobstmann, Michael Kishinevsky, and Jens Brandt, editors, *MEMOCODE*, pages 119–129. IEEE, 2011.

[BLRV95] A. Bröckers, C. M. Lott, H. D. Rombach, and M. Verlage. MVP-L Language Report Version 2. Technical Report Technical Report Nr. 265/95, University of Kaiserslautern, 1995.

[BMR01] S. Bistarelli, U. Montanari, and F. Rossi. Semiring-Based Constraint Logic Programming: Syntax and Semantics. *ACM TOPLAS*, 23(1):1–29, 2001.

[BMSG+09] Yuriy Brun, Giovanna Marzo Serugendo, Cristina Gacek, Holger Giese, Holger Kienle, Marin Litoiu, Hausi Müller, Mauro Pezzè, and Mary Shaw. Software engineering for self-adaptive systems. chapter Engineering Self-Adaptive Systems through Feedback Loops, pages 48–70. Springer, Berlin, Heidelberg, 2009.

[BPG+04] Paolo Bresciani, Anna Perini, Paolo Giorgini, Fausto Giunchiglia, and John Mylopoulos. Tropos: An Agent-Oriented Software Development Methodology. *Autonomous Agents and Multi-Agent Systems*, 8(3):203–236, May 2004.

[Cor05] IBM Corporation. An Architectural Blueprint for Autonomic Computing. Technical report, IBM, 2005.

[CPZ11] G. Cabri, M. Puviani, and F. Zambonelli. Towards a Taxonomy of Adaptive Agent-based Collaboration Patterns for Autonomic Service Ensembles. In *Proc. of CTS*, pages 508–515. IEEE, May 2011.

[DEU] Distributed Systems Group, DEUS project homepage. `http://code.google.com/p/deus/`.

[DFLP11a] Rocco De Nicola, Gian Luigi Ferrari, Michele Loreti, and Rosario Pugliese. A Language-Based Approach to Autonomic Computing. In *Formal Methods for Components and Objects, 10th International Symposium, Revised Selected Papers*, pages 25–48, 2011.

[DFLP11b] Rocco De Nicola, Gian Luigi Ferrari, Michele Loreti, and Rosario Pugliese. A Language-Based Approach to Autonomic Computing. In *Revised Selected Papers of FMCO*, pages 25–48. Springer, 2011.

[dLea11] Rogerio de Lemos et al. Software Engineering for Self-Adaptive Systems: A second Research Roadmap. In Rogerio de Lemos, Holger Giese, Hausi Müller, and Mary Shaw, editors, *Software Engineering for Self-Adaptive Systems*, number 10431 in Dagstuhl Seminar Proceedings, Dagstuhl, Germany, 2011. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, Germany.

[DLPT13] Rocco De Nicola, Michele Loreti, Rosario Pugliese, and Francesco Tiezzi. SCEL: a Language for Autonomic Computing. Technical report, IMT Lucca, January 2013.

[DNFP98] R. De Nicola, G.L. Ferrari, and R. Pugliese. KLAIM: A Kernel Language for Agents Interaction and Mobility. *Software Engineering, IEEE Transactions on*, 24(5):315–330, 1998.

[GSR96] Georg Gottlob, Michael Schrefl, and Brigitte Röck. Extending object-oriented systems with roles. *ACM Trans. Inf. Syst.*, 14(3):268–296, July 1996.

[Hel10] Joseph L. Hellerstein. Google cluster data. Google research blog, January 2010. Posted at `http://googleresearch.blogspot.com/2010/01/google-cluster-data.html`.

[HK14] Rolf Hennicker and Annabelle Klarl. Foundations for Ensemble Modeling - The Helena Approach. In *Specification, Algebra, and Software (Festschrift in Honour of Kokichi Futatsugi)*, 2014. To Appear.

[Höl13] Matthias Hölzl. The POEM Language (Version 2). Technical Report 7, ASCENS, July 2013. `http://www.poem-lang.de/documentation/TR7.pdf`.

[HW11] Matthias M. Hölzl and Martin Wirsing. Towards a system model for ensembles. In Gul Agha, Olivier Danvy, and José Meseguer, editors, *Formal Modeling: Actors, Open Systems, Biological Systems*, volume 7000 of *Lecture Notes in Computer Science*, pages 241–261. Springer, 2011.

[HW14] Matthias Hölzl and Martin Wirsing. Issues in engineering self-aware and self-expressive ensembles. In Jeremy Pitt, editor, *The Computer After Me*. World Scientific Publishing, to appear 2014.

[HZWS12] N. Hoch, K. Zemmer, B. Werther, and R. Y. Siegwarty. Electric Vehicle Travel Optimization - Customer Satisfaction Despite Resource Constraints. In *Proc. of IEEE IVS*. IEEE, 2012.

[IM10] Paola Inverardi and Marco Mori. A Software Lifecycle Process to Support Consistent Evolutions. In Rogério de Lemos, Holger Giese, Hausi A. Müller, and Mary Shaw, editors, *Software Engineering for Self-Adaptive Systems*, volume 7475 of *LNCS*, pages 239–264. Springer, 2010.

[JL87] J. Jaffar and J. L. Lassez. Constraint Logic Programming. In *POPL*, pages 111–119. ACM Press, 1987.

[KBP+13a] Jaroslav Keznikl, Tomas Bures, Frantisek Plasil, Ilias Gerostathopoulos, Petr Hnetynka, and Nicklas Hoch. Design of Ensemble-Based Component Systems by Invariant Refinement. In *Proceedings of the 16th International ACM Sigsoft symposium on Component-based software engineering*, CBSE '13, pages 91–100, New York, NY, USA, 2013. ACM.

[KBP+13b] Jaroslav Keznikl, Tomas Bures, Frantisek Plasil, Ilias Gerostathopoulos, Petr Hnetynka, and Nicklas Hoch. Design of Ensemble-Based Component Systems by Invariant Refinement. In *Proc. of CBSE '13*, Vancouver, Canada, 2013. ACM.

[KC03] Jeffrey Kephart and David Chess. The Vision of Autonomic Computing. *Computer*, 36(1):41–50, 2003.

[Lam08] Axel Van Lamsweerde. Requirements Engineering: from Craft to Discipline. In *SIGSOFT '08/FSE-16*, pages 238–249. ACM, 2008.

[MCY99] John Mylopoulos, Lawrence Chung, and Eric S. K. Yu. From Object-Oriented to Goal-Oriented Requirements Analysis. *Communications of the ACM*, 42(1):31–37, 1999.

[MES] MESSI: `http://sysma.lab.imtlucca.it/tools/ensembles/`.

[MMH12] G. V. Monreale, U. Montanari, and N. Hoch. Soft Constraint Logic Programming for Electric Vehicle Travel Optimization. *CoRR*, abs/1212.2056, 2012.

[MVZ+12] Lukáš Marek, Alex Villazón, Yudi Zheng, Danilo Ansaloni, Walter Binder, and Zhengwei Qi. DiSL: a domain-specific language for bytecode instrumentation. In *Proceedings of the 11th annual international conference on Aspect-oriented Software Development*, AOSD '12, pages 239–250, New York, NY, USA, 2012. ACM.

[NFLP13] Rocco Nicola, Gianluigi Ferrari, Michele Loreti, and Rosario Pugliese. A language-based approach to autonomic computing. In Bernhard Beckert, Ferruccio Damiani, FrankS. Boer, and MarcelloM. Bonsangue, editors, *Formal Methods for Components and Objects*, volume 7542 of *Lecture Notes in Computer Science*, pages 25–48. Springer Berlin Heidelberg, 2013.

[NGT04] Dana Nau, Malik Ghallab, and Paolo Traverso. *Automated Planning: Theory & Practice*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2004.

[OGCD10] Rehan O'Grady, Roderich Groß, Anders Lyhne Christensen, and Marco Dorigo. Self-assembly strategies in a group of autonomous mobile robots. *Autonomous Robots*, 28(4):439–455, 2010.

[PCZ13] M. Puviani, G. Cabri, and F. Zambonelli. A taxonomy of architectural patterns for self-adaptive systems. In *Proceedings of the Sixth International C* Conference on Computer Science and Software Engineering*, pages 76–84, Porto, Portugal, July 2013.

[PF13] Mariachiara Puviani and Regina Frei. Self-management for cloud computing. In *SAI Conference, London, UK*, 2013.

[PNA+13] Mariachiara Puviani, Victor Noel, Dhaminda Abeywickrama, Franco Zambonelli, Rocco De Nicola, Francesco Tiezzi, Luca Cesari, and Rosario Pugliese. Third report on wp4. *ASCENS Deliverable D*, 4, 2013.

[PPC+13] M. Puviani, C. Pinciroli, G. Cabri, L. Leonardi, and F. Zambonelli. Is self-expression useful? evaluation by a case study. In *Proceedings of the 22nd IEEE WETICE conference - 3rd Track on Adaptive and Reconfigurable Service-oriented and conmponent-based Applications and Architectures (AROSA)*, Hammamet, Tunisia, June 2013.

[PTO+12] Carlo Pinciroli, Vito Trianni, Rehan O'Grady, Giovanni Pini, Arne Brutschy, Manuele Brambilla, Nithin Mathews, Eliseo Ferrante, Gianni Di Caro, Frederick Ducatelle, Mauro Birattari, Luca Maria Gambardella, and Marco Dorigo. ARGoS: a modular, parallel, multi-engine simulator for multi-robot systems. *Swarm Intelligence*, 6(4):271–295, 2012.

[Puv12] M. Puviani. Tr 4.2: Catalogue of architectural adaptation patterns. Technical report, ASCENS Project, 2012.

[Puv13] M. Puviani. Tr 4.3: Simulation of adaptation patterns and self-expression mechanisms. Technical report, ASCENS Project, 2013.

[RRMP08] Andreas Rausch, Ralf Reussner, Raffaela Mirandola, and Frantisek Plasil, editors. *The Common Component Modeling Example: Comparing Software Component Models*, volume 5153 of *LNCS*. Springer, 2008.

[Ser13] Nikola Serbedzija. Deliverable d7.3: Third report on wp7 - integration and simulation report for the ascens case studies. Technical report, ASCENS Project, 2013.

[SF09] Nikola B. Serbedzija and Stephen H. Fairclough. Biocybernetic loop: From awareness to evolution. In *IEEE Congress on Evolutionary Computation*, pages 2063–2069. IEEE, 2009.

[ST09] M. Salehie and L. Tahvildari. Self-adaptive software: Landscape and research challenges. *ACM TAAS*, 4(2):1–42, 2009.

[Szy02] Clemens Szyperski. *Component Software: Beyond Object-Oriented Programming*. Addison-Wesley, Boston, MA, USA, 2nd edition, 2002.

[VH13] E. Vassev and M. Hinchey. Autonomy Requirements Engineering. *IEEE Computer*, 46(8):82–84, 2013.

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

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