The Ensemble Development Life Cycle and Best Practises for Collective Autonomic Systems *

Matthias Hölzl[1], Nora Koch[1], Mariachiara Puviani[2], Martin Wirsing[1], and Franco Zambonelli[2]

[1] Ludwig Maximilians-Universität München
[2] University of Modena and Reggio Emilia, Italy

Abstract. Collective autonomic systems are adaptive, open-ended, highly parallel, interactive and distributed software systems. Their key features are so-called self-* properties, such as self-awareness, self-adaptation, self-expression, self-healing and self-management. We propose a software development life cycle that helps developers to engineer adaptive behavior and to address the issues posed by the diversity of self-* properties. The life cycle is characterized by three feedback loops, i.e. based on verification at design time, based on monitoring and awareness in the runtime, and the feedback provided by 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. In addition, a pattern catalog for the development of collective autonomic systems is presented to ease the engineering process.

Keywords: software development life cycle, patterns, ensembles, awareness, adaptation, autonomic systems

1 Introduction

Software is increasingly used to model or control massively distributed and dynamic collective autonomic systems. These systems consist of a set of usually open-ended, highly parallel and interactive components, which operate in highly variable, even unpredictable, environments. Their key features are so-called self-* properties, such as self-awareness, self-adaptation, self-expression, self-healing and self-management.

Self-awareness is concerned with knowledge the system has about the system's behavior and the environment, which may be centrally held or distributed in nature. However, in designing autonomic self-aware systems, it is useful to explicitly and separately consider the process of determining systems actions as a

* This work has been sponsored by the EU project ASCENS IP 257414 (FP7).

result of this knowledge. This process is called self-adaptation or self-expression. In particular, if the autonomic system is recovering from some failure, the term self-healing is used. We distinguish also a self-management property as the ability collective autonomic systems have to manage local and global knowledge in order to be aware of their own state and the state of the environment. The knowledge is used for reasoning, learning and adapting at runtime to the system's and environmental changes.

One of the main challenges for software engineers is then to find reliable methods and tools to build the software that implement those self-* features required by collective autonomic systems. The main aim of the ASCENS project[3] is to tackle these issues using an engineering approach based on service components and ensembles. Ensembles are dynamic groups of components that are formed on demand to fulfill specific goals by taking into account the state of the (changing) environment they are operating in. One distinguishing characteristic of the approach is the use of formal methods to guarantee that the behavior of the software complies to the specifications.

In this chapter we present the Ensemble Development Life Cycle (EDLC) that covers the full design and runtime aspects of collective 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 control loops: *(1)* at design time, *(2)* at runtime and *(3)* between the two of them allowing for the system's evolution. The *design feedback control loop* enables continuous improvement of models and code due to changing requirements and results of verification or validation. The *runtime feedback control loop* implements self-adaptation based on awareness about the system and its environment. Finally, the *evolution feedback control loop* provides the mechanisms to change architectural models and code on the basis of the runtime behavior of the continuous evolving system.

We illustrate the EDLC using methods and tools, mostly developed within the ASCENS project. Examples are SOTA [2] for requirements engineering of awareness and adaptive issues, SCEL ([31], Chapter I.1 [53]) for modeling and programming, SBIP ([9], Chapter I.3 [28]) for verification, SPL ([20], Chapter II.5 [18]) for monitoring, Iliad (Chapter II.4 [41]) as awareness-engine, and JDEECo [22] and JRESP (Chapter I.1 [53]) as runtime frameworks. These methods and tools are specifically designed to capture the self-* features of autonomic systems.

When complex collective autonomic systems are developed, an important aspect of the development process is the reusability of design choices, i.e. the advantage to identify architectural schemes that can be reused at component and ensemble level. We have defined a *pattern catalog* of interrelated patterns –a so-called *pattern language*. Such a pattern language enables developers to choose different design elements making the resulting models, implementation and selected verification techniques more understandable. We illustrate the pat-

[3] ASCENS website: http://www.ascens-ist.eu/

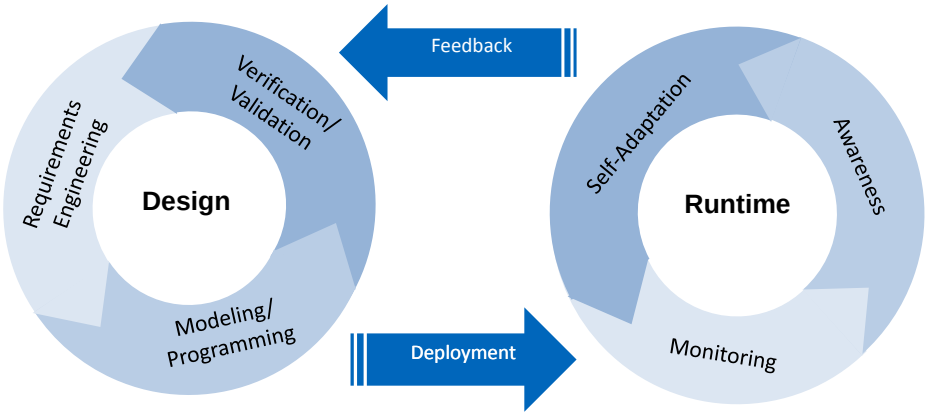

Fig. 1. Ensembles Development Life Cycle (EDLC).

tern catalog with a set of architectural patterns focusing mainly on the feedback control loops they support.

The structure of the chapter is as follows: Section II provides an overview of the EDLC. Section III focuses on the feedback control loops and their relationship to the phases of the EDLC. Section IV present a pattern language for ensemble development and a set of pattern examples. Section V relates our work to other relevant software engineering approaches, and Section VI concludes with a summary and future challenges regarding the engineering of collective autonomic systems.

2 Software Development Life Cycle for Ensembles

The development of collective autonomic systems goes beyond addressing the classical phases of the software development life cycle like requirements elicitation, modeling, implementation and deployment. Engineering these complex systems has also to tackle aspects such as self-* properties 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. Therefore, we need to capture how the system should be adapted and how the system or environment should be monitored in order to make adaptation possible.

Models are usually built on top of the elicited requirements, mainly 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. A relevant issue is then the use of modeling and implementation techniques for adaptive and awareness features. Our aim is to focus on these distinguishing characteristics of collective autonomic systems along the whole development cycle.

We propose a "double-wheel" life cycle 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 mathematically founded approaches to validate and verify the properties of the collective autonomic system and enable the prediction of the behavior of such complex software. 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 running system and the environment, reasoning on such observations and using the results of the analysis for adapting the system and providing feedback that can be used in the offline activities.

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

The process defined by the EDLC can be refined by providing details on the involved stakeholders, the methods and tools they will use in the development as well as the input needed and the output produced in each stage. This will ease the selection of the most appropriate tools for each collective autonomic system to be build. Process modeling languages can be used to specify these details: Either general workflow-oriented modeling languages such as UML activity diagrams[4], and BPMN[5], or a Domain Specific Language (DSL) such as the OMG standard Software and Systems Process Engineering Metamodel (SPEM)[6] and the Multi-View Process Modeling Language (MV-PML) developed by NASA [13].

Figure 2 shows an example of a process model specified in SPEM for the requirements engineering steps of the e-Mobility scenario described in [24]. It illustrates the relationships between stakeholders like the requirements engineer, actions such as the definition of adaptation goals and process inputs like interviews and results such as the SOTA model and the IRM model. Aspects of the runtime phases of the development process of this scenario are shown in Fig. 3 focusing on the monitoring and adaptation activities that use JDEECo components and enables feedback to the phases of the design "wheel".

3 Engineering Feedback Control Loops

Feedback control loops are the heart of any collective autonomic system providing the generic mechanism for adaptation and enabling the creation of flexible runtime solutions by monitoring the subsystem and applying corrections in order to approach the goals. Moreover they allow a system to become self-aware

[4] UML website: http://www.uml.org/
[5] BPMN website: http://www.omg.org/spec/BPMN/2.0/
[6] SPEM website: http://www.omg.org/spec/SPEM/2.0/

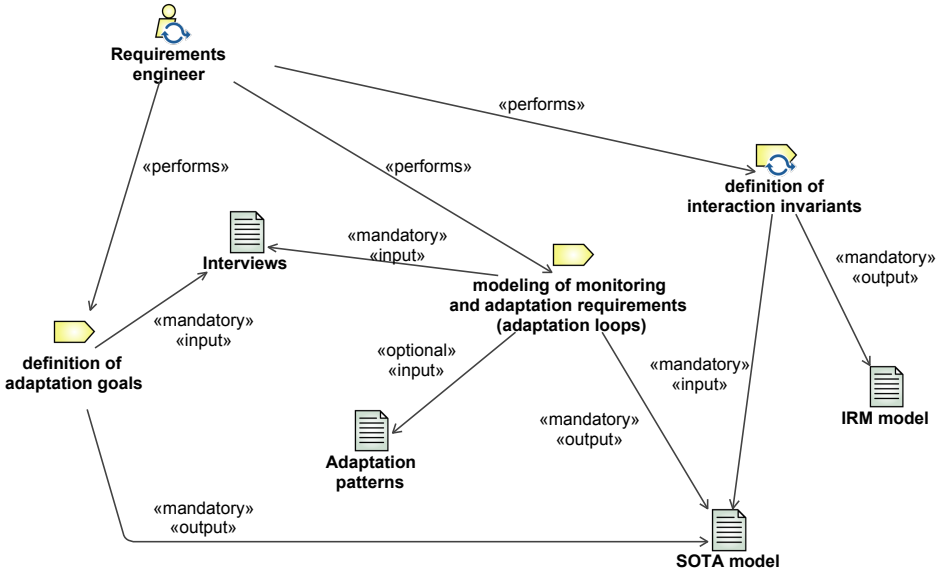

Fig. 2. E-Mobility Scenario Development Process: Requirements Engineering Aspects.

with respect to the quality of its operations, and self-healing if there are any problems. This is achieved by approaches like MAPE-K [30] that collect appropriate runtime data and analyzing it, and by planning and executing adaptation strategies.

Engineering approaches for building collective autonomic systems need to consider feedback control loops from the beginning on and all over the development life cycle. This includes requirements that make such loops necessary, the influence loops have on architecture, deployment aspects to be taken into account, additional features to be supported as monitoring and awareness of the system's and environmental behavior, and implementation of adaptation mechanisms.

These engineering features are considered by the EDLC presented in the previous section, which itself is composed of three feedback loops: at design time, at runtime, and from runtime back to improve the design with the associated redeployment of the evolving software. The development of collective autonomic system offers several engineering challenges that are addressed by the methods, techniques and tools that have been developed in the ASCENS project. The remainder of this section focus on the feedback loops and provides examples of ASCENS approaches that can be used in the different phases of the development life cycle within each feedback loop.

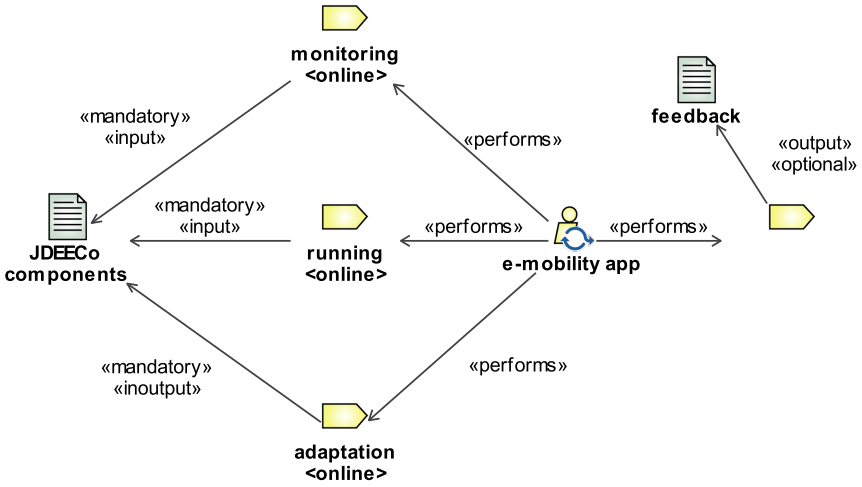

Fig. 3. E-Mobility Scenario Development Process: Runtime Aspects.

3.1 Design Cycle

The "design wheel" comprises three phases: *requirements engineering, modeling and programming, and verification and validation*. At first glance it resembles a traditional software development life cycle. However, an autonomic ensemble, has to be aware of other autonomic ensembles and take into account its environment. It has to provide and consume knowledge, manage policies specified in form of goals, rules and/or constraints and infer lower-level actions [45]. These features of collective autonomic systems have to be addressed in the stages prior to *programming*, too, i.e. *requirements engineering* and *modeling*. In particular, goal-oriented approaches are in these stages promising techniques used for requirements elicitation and specification. Particularly challenging are also the *validation and verification* of large-scale autonomic systems as it will be harder to anticipate their environment.

Requirements Engineering Traditionally, software engineering divides requirements in two categories: functional requirements (what the system should do) and non-functional requirements (how the system stands for achieving its functional requirements in terms of performance, quality of service, etc.). In the area of adaptive systems, and more in general of open-ended systems, both functional and non-functional requirements are better expressed in terms of "goals" [52]. A goal, in most general terms, represents a desirable state of the affairs that a software component or software system, aims to achieve. In fact, a self-adaptive system/component should be engineered not simply to "achieve" a functionality or state of the affairs, but rather to "strive to achieve" such functionality per-

haps in several steps, i.e., be able to take self-adaptive decisions and actions so as to preserve its capability of achieving despite contingencies.

Within the ASCENS project a couple of different *goal-oriented* approaches were proposed to elicitate and specify the requirements of collective autonomic systems, i.e. State of the Affairs (SOTA), General Ensemble Model (GEM), Invariant Refinement Method (IRM) and Autonomy Requirements Engineering (ARE) that we briefly sketch below.

The SOTA approach [2] proposes an extension of existing goal-oriented requirements engineering approach that integrates elements of dynamic systems modeling. SOTA models the entities of a self-adaptive system as if they were immersed in n-dimensional space $\mathbf{S}$, each of the n dimensions of such space representing a specific aspect of the current situation of an entity/ensemble and of its operational environment. As the entity executes, its position in $\mathbf{S}$ changes either due to its specific actions or because of the dynamics of environment. Thus, we can generally see this evolution of the system as a movement in $\mathbf{S}$. For example, in the ASCENS e-mobility scenario described in [24], the space $\mathbf{S}$ includes the spatial dimensions related to the street map, but also dimensions related to the current traffic conditions, the battery conditions, etc. Once the SOTA space is defined, a goal is specified in SOTA; 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.

Along these 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 get activated and any possible constraint on the trajectory to be followed while trying to achieve such goals.

The General Ensemble Model (GEM) is a mathematical formalization of the SOTA approach that gives precise semantics to SOTA models and provides ways to specify model properties in various logics [43], such as the higher-order logic of the PVS system [54] or various temporal logics. The precise semantics that GEM provides for SOTA models enables developers to analyze requirements models using mathematical techniques; Chapter 1.3.1 in this volume shows how a SOTA/GEM model may be used to derive an adaptation strategy for a swarm of robots operating in an adversarial environment by applying concepts from (differential and evolutionary) game theory.

The Autonomy Requirements Engineering (ARE) (described in detail in Chapter III.3 [58]) of this volume uses a goal-oriented approach, too, along with a special model for generic autonomy requirements (GAR). ARE starts with the creation of a goals model that represents system objectives and their

inter-relationships. In the next phase, the GAR model is used to refine each one of the system goals with elicited environmental constraints to come up with self-* objectives providing autonomy requirements for achieving these goals. The autonomy requirements are derived in the form of goal-supportive and alternative self-* objectives, along with required capabilities and quality characteristics. Finally, these autonomy requirements can be specified with KnowLang, a framework dedicated to knowledge representation for self-adaptive systems.

For the refinement of the high-level strategic goals defined in SOTA to the architecture of the collective autonomic system in terms of low-level components and ensemble, we can use the Invariant Refinement Method (IRM) [46] (see also Chapter III.4 [23]of this volume). The main idea of IRM is to capture the high-level goals and requirements in terms of invariants, which describe the desired state of the system-to-be at every time instant. Invariants are to be maintained by the coordination of the different system components. As a design decision, top-level invariants are iteratively decomposed into more concrete sub-invariants, forming a decomposition graph with traceability of design decisions.

The IRM approach has been used e.g. to model the functional and adaptive requirements of the e-Mobility scenario, capturing the necessity to keep the vehicles plan updated and to check whether the current plan remains feasible with respect to measured battery level. The identified leaf invariants are easily mappable to component activities, which are further formally specified by (SCEL) [32] or SCPL [21] (see Sec. 3.1).

The requirements engineering approaches SOTA, GEM, ARE and IRM complement each other and are useful to understand and model the functional and adaptation requirements, and to check the correctness of such specifications (as described in [1]). However, when a designer considers the actual design of collective autonomic system, it is necessary to identify which architectural schemes need to be chosen for the individual components and the ensembles. In the next section we give an overview of the taxonomy of architectural patterns we defined [26] for adaptive components and ensemble.

Modeling and Programming The Service Component Ensemble Language (SCEL) [32, 31] has been designed to deal with adaptation and move toward the actual implementation of the self-* properties identified during the requirements engineering phase, which have been specified e.g. using the IRM approach. 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, are equipped with an interface, a knowledge repository, a set of policies, and a process.

The language is equipped with an operational semantics that permits verification of formal properties of systems. Moreover, a SCEL program can rely on separate reasoning components that can be invoked when decisions have to be taken.

To move toward implementation, jRESP[7], 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. It is worth noticing that the implementation of jRESP fully relies on the SCEL's formal semantics. This close correspondence enhances confidence in the behavior of the jRESP implementation of SCEL programs, once the latter have been analysed through formal methods made possible by the formal operational semantics. For more details on SCEL and jRESP the reader is referred to Chapter I.1 [53] of this volume.

As a complement to SCEL, within the ASCENS context an approach called Soft Constraint Logic Programming (SCPL) (see Chapter II.5 [18]) has been proposed and applied to an e-Mobility travel optimization scenario. Besides optimizing trips of single users, so-called local problems, the e-Mobility case study aims to solve global problems involving large ensembles of vehicles. To tackle these a coordination of declarative and procedural knowledge is used and a decomposition of the global problem into local problems, which are solved by SCLP implementations and whose parameters can be iteratively determined by a programmable orchestration.

Complementary approaches to SCEL, like Helena [48] and DEECo, have been developed for the specification of collective autonomic systems as well within the scope of ASCENS. The Helena approach proposes a role-based method for modeling collaborations using a UML-like notation and is founded on a rigorous formal semantics. Helena focuses on the description of the behavior of each role as well as on the behavior on the ensemble level.

DEECo (Dependable Emergent Ensembles of Components) component model [47, 22] can be used to provide us with the relevant software engineering abstractions that ease the programmers tasks. A component in DEECo, features an execution model based on the MAPE-K [30] autonomic loop. 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 realtime manner.

In order to bring the above abstractions to practical use we have used jDEECo our reification of DEECo component model in Java. In jDEECo, com-

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

ponents 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.

Verification and Validation When dealing with complex collective 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. These systems are deployed for increasingly complex tasks; and it becomes more and more important to prove as early as possible in the development life cycle that they are safe, dependable, and correct.

Analysis techniques for collective autonomic systems that capture essential aspects such as adaptive behavior, interactions between the components, and changing environments can only succeed if they exploit as much as possible the specific structure of the considered systems (e.g. large replication of the same component, hierarchical compositions). In ASCENS we consider both, *qualitative analyses* targeting boolean properties stating that the system behaves without any flaw, and *quantitative analyses* that evaluate expected performances according to predefined metrics (energy/memory consumption, average/maximum time to accomplish a task, probability to fulfil a goal, etc.). We also address security specific issues such as control policies and information flow.

Our approach for dealing with *qualitative properties* is to use so called formal verification techniques, which provide a mathematical proof, e.g. model-checking and theorem prover. Formal verification is an attractive alternative to traditional methods of testing and simulation that can be used to provide correctness guarantees. Hereby, we mean not just the traditional notion of program verification, where the correctness of code is at question. We more broadly focus on design verification, where an abstract model of a system is checked for desired behavioral properties. Finding a bug in a design is more cost-effective than finding the manifestation of the design flow in the code.

Regarding *quantitative properties* and system performance, the environment in which the system is immersed plays an important role. Therefore the environment behavior has to be modeled as well providing additional information on possible scenarios the system may present. In ASCENS we used stochastic models and frameworks for the evaluation of quantitative properties related to the case studies of the project.

Considering *security aspects*, the focus was on confidentiality issues. We develop a model-driven framework for information flow analysis, named *secBIP* [7], which is suited for checking non-interference, a system property stating that information about higher security levels cannot be inferred from lower security levels. This component-based framework allows for the construction of complex systems by composition of atomic components with communication and coordination operators.

Several tools have been implemented within ASCENS to support these verification and validation methods, some of them as extension of well known existing tools. We mention here the most relevant: D-Finder [10, 11], SMC-BIP [6], SBIP [9] and GMC.

The first three are based on BIP, a formal framework for building heterogeneous and complex component-based systems [8]. Notably, thanks to the formal operational semantics of the SCEL language, 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. D-Finder is a tool used for the compositional verification of safety properties, that is, it aims at producing proofs stating that ensemble of components cannot reach a predefined set of undesirable states. The method developed combines structural analysis for component behaviors with structural analysis of connectors. SMC-BIP is a tool to perform quantitative analysis, using formally-defined models from which it explores the reachable states. Its main characteristic is to provide answers to quantitative questions based on partial state-space coverage. It also evaluates confidence in such results based on stochastic models. SBIP is an extension of BIP that allows stochastic modeling and statistical verification. On one hand, it relies on BIP expressiveness to handle heterogeneous and complex component-based systems. On the other hand it uses statistical model checking techniques to perform quantitative verification targeting non-functional properties. GMC is a model checker that verifies whether properties of service components are satisfied by their implementations in the C or C++ language, i.e. that in any thread interleaving, no deadlock appears and no assertion state in the code is violated. It supports verification of multi-threaded programs.

For details on verification and validation methods and tools for collective autonomic systems, the reader is referred to Chapter I.3 [28] of this volume.

3.2 Runtime Cycle

The "runtime wheel" comprises the online activities the system performs autonomically: *monitoring, awareness and adaptation*. This cycle is characteristic for the life cycle of collective autonomic systems and a major difference from traditional software which has a much more static runtime behavior.

The runtime collecting of data about the system, its components or its environment is called *monitoring*. Monitoring is an essential feature of adaptive systems. While sometimes systems react directly to the data obtained by the monitor, it is more common for ensembles to pass this data to an *awareness mechanism*, i.e., a subsystem that contains reasoners, planners or learning mechanisms, that allow an autonomic system to come up with responses to the challenges posed by its environment. *Adaptation*, in this context, is then the act of implementing the decisions made by the awareness mechanism, e.g., by performing different actions done before or by reconfiguring the system structure. To manage their collective behavior, self-aware components in an ensemble may need to communicate with each other. Therefore the phases of the runtime cycle are not restricted to actions taken by a single component, they may also

involve the joint activities of multiple components. For example, when a robot in a swarm becomes aware of a danger to the swarm it should communicate the presence of the danger to other robots in the swarm.

Monitoring Monitoring is the first step in the runtime cycle of any adaptive system: without information about the state of the system or environment, any change in behavior can only be a random activity and not a purposeful action of the system. Individual components, subsystems of a collective autonomic system, the whole system or parts of the environment may all be monitored.

In the double-wheel life cycle, monitoring has a dual role. The primary objective is usually to provide information about the current state of the system and its environment to the awareness mechanism, which incorporates this information into the decision making process. A secondary objective is to provide developers feedback about properties of the environment so that they can check whether the behavior of the awareness mechanism is adequate and achieves the desired goals.

One of the technical challenges faced by monitoring systems is *dynamic coverage configuration*: The awareness mechanism may require different information at different points. Monitoring should accommodate these 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 awareness benefit provided by the data. Often *high monitoring coverage* is necessary to accommodate the requirements of the awareness mechanism, but sometimes this may lead to monitoring costs that are higher than the benefits gained by the additional situational awareness.

To support easy access to monitoring perfomance information in ASCENS, we have developed SPL [20], 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 [49]. In [19] and the Chapter II.5 [18] of this volume we explain how the two technologies interact in the context of a performance-aware component system.

Awareness The knowledge a collective autonomic system has on its behavior and environment as well as the reasoning mechanisms that can be employed by the system at runtime comprise its awareness. We divide the notion of awareness along four dimensions: *scope* (which parts of the system and environment are represented by the awareness mechanism), *breadth* (which properties are part of the awareness model), *depth* (which kinds of questions the awareness mechanism can answer) and *quality* (how well the conclusions the ensemble draws correspond to reality). Chapter II.4 [41] contains a more detailed discussion of awareness mechanisms.

To enable problem solving and adaptation in complex domains, *deep awareness mechanisms* may be required. Deep models and reasoners can not only an-

swer questions about the immediately observable state of the system, they also model underlying principles such as causality or physical properties so that they may, e.g., infer consequences of actions or diagnose likely causes of unexpected events.

Designers cannot provide a complete specification of the conditions in which an autonomic system has to operate. To achieve the desired performance and to allow flexible reactions to contingencies not foreseen at design-time, the awareness mechanism may need to learn how to adapt its internal models to the circumstances encountered at runtime.

The POEM language [40] 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*, i.e., programs in which certain operations are left as non-deterministic choices for the runtime 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 statistically 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.

The Iliad implementation of POEM includes facilities for full first-order inference and special-purpose reasoners for, e.g., temporal and spatial reasoning; their results can be combined with planning methods to compute long-term strategies if enough information about the ensemble's operating conditions is available. In addition, it can compute completions of programs using hierarchical-reinforcement-learning techniques. Iliad is fully integrated as knowledge repository and reasoner in jRESP and can therefore be used as awareness engine for SCEL programs.

Self-adaptation Once the awareness mechanism of a component or ensemble has come to the conclusion that a malfunction, contingency, or simply a performance issue exists, it has to decide how to respond in order to resolve the situation.

In ASCENS we call this response an adaptation action, and we distinguish between two main classes of adaptation actions:

- *Weak-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.
- *Strong-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.

Weak adaptations are often cheaper and simpler than strong adaptations and still sufficient to respond adequately to changes in its environment: If the path of a rescue robot is blocked it can simply try to take another route to its target; there is no need for the robot to change its configuration to respond to

this scenario. However, for more difficult adaptations, the whole structure of an ensemble may need to be reconfigured: If a swarm of independently operating rescue robots has to move victims that are too heavy for a single robot to carry, several robots may have to form a new sub-ensemble that coordinates their actions using a centralized autonomic manager. The adaptation patterns presented in Sect. 4 support these kinds of strong adaptation.

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. For white-box and black-box adaptation mechanisms for collective autonomic systems the reader is referred to Chapter II.1 [16] of this volume.

3.3 Evolution Control Loop

The two cycles of EDLC are complemented by transitions from design cycle to runtime cycle and vice versa supporting the long term system's evolution. The collective autonomic system evolution consists in monitoring data at runtime, the fed back to the design cycle to provided basis for system redesign and redeployment. These transitions thus correspond to *deployment* and *feedback* activities.

Deployment The transition from design to runtime deploys a collective autonomic system preparing it for its execution. 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 like JDEECo [22] and JRESP. These frameworks allow for the distributed execution of a service component application and provide their specific means of deployment.

Feedback The transition from runtime to design provides feedback based on data collected by the monitoring and learning process of the running application. The feedback is used for improving the specification, code or a deeper analysis of the requirements. It connects the online with the offline development process. This connection is made possible by employing design methods that keep the traceability of design decisions to code artefacts and knowledge – e.g. the Invariant Refinement Method (IRM) [46], which has been specifically developed for hierarchical design of a service component application. When used in conjunction with IRM, monitoring *(i)* observes the real functional and non-functional properties of components and situation in components environment, and *(ii)* provides observed data. At design time these observed data can be compared to assumptions and conclusions captured by IRM; comparison is currently performed manually but we envision automated support. 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. For further details on IRM, see Chapter III.4 [23] of this volume.

4 A Pattern Language for Ensemble Development

In order to design and develop collective autonomic systems, we have defined a catalog of patterns for this kind of systems [55, 42]. The importance of the catalog and of patterns in general start from the idea that "software patterns are reusable solutions to recurring design problems and are considered a mainstream of software reuse practice" [51]. So software adaptation can indeed benefit from reuse in a similar way that designing software architectures has benefited from the reuse of software design patterns [37].

Presenting engineering concepts in terms of interrelated patterns enables developers to explore the relationship between different design elements and simplifies an understanding of the trade-offs involved in different modeling, verification and implementation choices. To support the full development life cycle and to be usable for developers who are not already expert in the EDLC and the various technologies developed by ASCENS we have included patterns at different levels of abstraction so that the pattern catalog [39] can also serve as introduction to certain development techniques.

4.1 Pattern Categories

We started identifying adaptation patterns, one of the undoubtedly most important and unique design aspects of ensembles. There are, however, many other parts of the ensemble development process where interrelated design challenges and implementation choices can be clarified and made accessible via a catalog of interrelated patterns, which is often called a pattern language.

Currently our pattern catalog contains patterns in the following areas:

Conceptual Patterns: High-level descriptions of certain techniques or concepts that can serve as introduction to topics with which developers may not be familiar. An example is *Awareness Mechanism* that describes the general concept of those mechanisms to ensure awareness of the system's and environmental behavior.

Architectural Patterns: Patterns that describe the architecture of a system or a component. An example for a pattern in this category is *Distributed Awareness-based Behavior*. These patterns often serve as entry points into the catalog for developers trying to solve an architectural problem.

Adaptation Patterns: Patterns concerned with adaptation and the control-loop structure of ensembles. Examples for patterns in this area are *Reactive Stigmergy Service Components Ensemble Pattern* and *Centralised AM Service Components Ensemble Pattern* described in detail in section 4.3.

Awareness Patterns: Patterns for developing and using awareness mechanisms. An example is *Action-calculus Reasoning*, a pattern that describes the trade-offs in using a logical formalism based on an action calculus for modeling and reasoning about the system's domain.

Coordination Patterns: Patterns that are concerned with coordination aspects of an ensemble. An example for a pattern in this category is *Tuple-space Based Coordination*.

Cooperation Patterns: Patterns that describe mechanisms for cooperation between agents in an ensemble. For example the *Auction* mechanism belongs to this category.

Implementation Patterns: Patterns that are mainly concerned with implementation or low-level design aspects. An example is the *Monkey Patching* (anti)-pattern which deals with a certain method of dynamic code update.

Knowledge Patterns: Patterns that addresses issues arising with the development of knowledge bases and knowledge-based systems. Examples for patterns in this category are *Build Small Ontology* or *Reuse Large Ontology*.

Navigation Patterns: Patterns that address navigation or position keeping in physical space, for example *Build Chain to Target*.

Self-expression Patterns: Patterns that are concerned with self-expression of ensembles, and goal-directed or utility-maximizing behaviors. A simple example is *Decompose Goal into Subgoals*.

These categories are neither exhaustive nor disjoint. Patterns such as *Cooperate to Reach Goal* belong into several categories (cooperation patterns and self-expression patterns), and it is easy to think of patterns which do not fit in any of the categories mentioned above. Therefore, the classification of patterns is done via keywords, which allow *m-n* relationships between patterns and categories and make it easy to introduce new categories. For each pattern that is concerned with particular phases of the EDLC, these phases are also represented as keywords for the pattern.

As the *Monkey Patching* example shows, the catalog also includes some patterns that describe widely used but potentially dangerous techniques, so-called anti-patterns. We think it is important to also include anti-patterns since there are often good reasons why an anti-pattern has become widely used. In many cases anti-patterns are good solutions for specialized problems which are regularly applied in situations in which they are unnecessary or in which better solutions exist (this is the case for the *Monkey Patching* pattern). Additionally, developers might not even know that a certain practice is considered an anti-pattern, and they might not be aware of superior alternatives, or of ways to mitigate the downside of using the anti-pattern.

When exploring the pattern catalog [39], the first two categories of patterns (conceptual patterns and architectural patterns) serve as good entry points into the pattern system; patterns in these categories provide a coherent overview of a general topic, and the tree of references starting from patterns in these categories transitively spans the whole pattern catalog.

4.2 Pattern Template

In the following paragraphs we describe the template that we use for our pattern language. Since the patterns in our pattern system range from conceptional patterns to implementation patterns, we include a relatively large number of fields, but we allow several of them to be left empty. In the following description, mandatory fields are marked with an asterisk. Except for conceptual patterns,

each pattern should either contain a *context* field or the two fields *motivation* and *applicability*, but it should not contain all three.

Name:* A descriptive name for the pattern, e.g., *Algorithmic Planning*.

Specializes: A pattern may inherit properties from another pattern but modify certain fields. In this case the parent pattern is included in the *specializes* field and the differences are described in the respective fields.

Classification:* The set of keywords that describes, e.g., to which phases of the EDLC the pattern applies.

Aliases: Other names by which this pattern is known.

Intent:* The purpose for this pattern, what does the pattern accomplish?

Summary: For patterns which have a very long description, a summary that addresses the most important features may be given in this field.

Context:* The design problem or runtime conditions to which this pattern is applicable. This field is mandatory for adaptation patterns; for other patterns the context is often split into motivation and applicability.

Motivation:* The reasons why this pattern is necessary.

Applicability:* Describes for which systems the pattern is applicable, and which influences might lead to other patterns being preferable.

Diagram/Structure: If applicable a diagram that describes the pattern; e.g., adaptation patterns contain a diagram illustrating the components that are involved in the feedback loops.

Description/Behavior:* A description of the pattern.

Formal Behavior: If applicable a more formal description of the pattern's behavior can be given in this section. For example, all adaptation patterns include a specification using the State-of-the-Affairs (SOTA) [2] notation of their behavior, which comprises the description of the patterns goals, constraints and utilities.

Consequences: Consequences and trade-offs for using the patterns. If this section is present it often summarizes trade-offs already mentioned in the *description* field.

Implementation: Implementation techniques and practical tips for realizing this pattern. This section also includes references to ASCENS tools that are helpful for implementing the pattern.

Variants: If a pattern has simple variations which are not significant enough to justify their own patterns they are mentioned here.

Related Patterns: Related patterns, e.g., patterns that specialize the current pattern, alternatives for the current pattern or patterns that are useful in the implementation of the current pattern.

Applications: References to applications that apply self-adaptation patterns in different real life scenarios. This is very important because the catalog has to be based on experiences and/or on some solid formal ground, and on a solid organization.

4.3 Pattern Examples

The pattern language described above provides a flexible structure in which many kinds of patterns can be conveniently expressed while still retaining enough commonality to build a coherent system of patterns.

To give a flavor of the patterns we present an excerpt of five patterns of the ASCENS pattern catalog. Due to space restrictions we omitted the section *applications* for some of the examples. The complete catalog is available on the web [39]. For a detailed description as long as the taxonomy table of the adaptation patterns the reader is referred to [55].

Pattern: *Reactive Stigmergy Service Components Ensemble Pattern*

- **Name**: Reactive Stigmergy Service Components Ensemble.
- **Classification**: service-components-ensemble, edlc-requirements-engineering
- **Intent**: There are a large amount of components that are not able to directly interact one to each other. The components simply react to the environment and sense the environment changes.
- **Context**: This pattern has to be adopted when:
  - there are a large amount of components acting together;
  - the components need to be simple components, without having a lot of knowledge;
  - the environment is frequently changing;
  - the components are not able to directly communicate one with the other.
- **Structure**: See Figure 4.

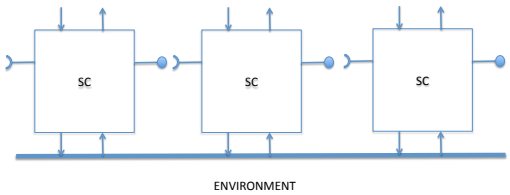

Fig. 4. Reactive Stigmergy Service Components Ensemble

- **Behavior**: This pattern has not a direct feedback loop. Each single component acts like a bio-inspired component (e.g. an ant). To satisfy its simple goal, the Service Component (SC) acts in the environment that senses with its "sensors" and reacts to the changes in it with its "effectors". The different components are not able to communicate one with the other, but are able to propagate information (their actions) in the environment. Than they are able to sense the environment changes (other components reactions) and adapt their behavior due to these changes.

– **SOTA description (Formal Behavior)**:
  - *Goals*: $G_{SC_1}, G_{SC_2}, \ldots, G_{SC_n}$
  - *Utilities*: $U_{SC_1} = U_{SC_2} = \ldots = U_{SC_n}$
  - *Explanation*: In the pattern each Service Component has a separated goal, that is explicit only at the component level.
    Regarding the utilities of the ensemble, they are the same of each SCs that have to be shared by the components.
– **Consequences**: If the component is a proactive one, its behavior is defined inside it with its internal goal. The behavior of the whole system cannot be a priori defined. It emerges from the collective behavior of the ensemble. The components do not require a large amount of knowledge. The reaction of each component is quick and does not need other managers because adaptation is propagated via environment. The interaction model is an entirely indirect one.
– **Related Patterns**: Proactive Service Component.

Pattern: *Centralised AM Service Components Ensemble Pattern*

– **Name**: Centralised Autonomic Manager (AM) Service Components Ensemble.
– **Classification**: service-components-ensemble, edlc-requirements-engineering
– **Intent**: A Service Component necessitates an external feedback loop to adapt. All the components need to share knowledge and the same adaptation logic, so they are managed by the same AM.
– **Context**: This patterns has to be adopted when:
  - the components are simple and an AM is necessary to manage adaptation;
  - a direct communication between components is necessary;
  - a centralised feedback loop is more suitable because a single AM has a global vision on the system;
  - there are few components composing the system.
– **Structure**: See Figure 5.
– **Behavior**: This pattern is designed around an unique feedback loop. All the components are managed by a unique AM that "control" all the components behavior and, sharing knowledge about all the components, is able to propagate adaptation.
– **SOTA description (Formal Behavior)**:
  - *Goals*: $G = G_{SC_1} + G_{SC_2} + \ldots + G_{SC_n} + G_{AM}$
  - *Utilities*: $U = U_{SC_1} + U_{SC_2} + \ldots + U_{SC_n} + U_{AM}$
– **Consequences**: An unique AM is more efficient to manage adaptation over the entire system, but it can became a node of failure.
– **Related Patterns**: Autonomic Service Component.

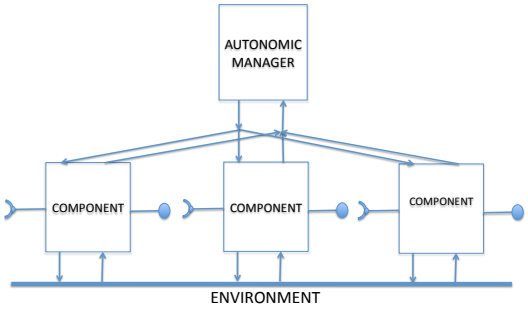

Fig. 5. Centralised AM Service Components Ensemble

Pattern: ***P2P AMs Service Components Ensemble Pattern***

- **Name**: P2P AMs Service Components Ensemble.
- **Classification**: service-components-ensemble, edlc-requirements-engineering
- **Intent**: This pattern is designed around an explicit autonomic feedback loop for each component. The components are able to communicate and coordinate each other through their AMs. Each AM manages adaptation on a single SC.
- **Context**: This pattern has to be adopted when:
  - the components are simple and an external AM is necessary to manage adaptation at the component level;
  - the components need to directly communicate one with the other (through their AMs) to propagate adaptation.
- **Structure**: See Figure 6.

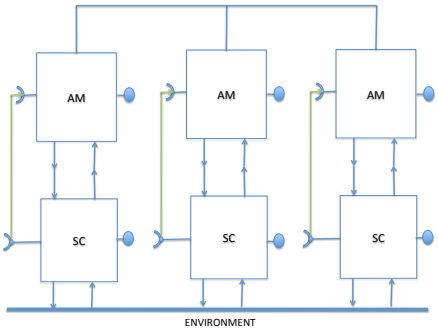

Fig. 6. P2P AMs Service Components Ensemble

– **Behavior**: Each component is managed by an AM and acts as an autonomic component. Than the AMs directly communicate one with the other with a P2P communication protocol. The communication made at the AM's level makes it easier to share not only knowledge about the components, but also the adaptation logic.
– **SOTA description (Formal Behavior)**:
  • *Goals*: $(G_{SC_1}, G_{AM_1}) \bigcup (G_{SC_2}, G_{AM_2}) \bigcup \ldots \bigcup (G_{SC_n}, G_{AM_n})$
  • *Utilities*: $(U_{SC_1}, U_{AM_1}) \bigcup (U_{SC_2}, U_{AM_2}) \bigcup \ldots \bigcup (U_{SC_n}, U_{AM_n})$

  The goal of the ensemble is composed of the goals of every single component. Here a component is composed of a SC and an AM, so its goal is the goal of the SC (if it is a proactive component), along with the goal of the AM.
  At the same way the utilities of the ensemble are composed of the utilities of every single component. In this scenario, it is not necessary that all the components have the same utilities (same for goals), and also some components may have no utilities at all.
– **Consequences**: The use of AMs to communicate between components makes the adaptation management more simple because the components remain simple and the knowledge necessary for adaptation is easily shared between the AMs.
– **Related Patterns**: Autonomic Service Component.
– **Applications**: A lot of case studies about intelligent transportation systems use this pattern. For example a traffic jam monitoring system case study is presented in [4]. The intelligent transportation system consists of a set of intelligent cameras, which are distributed evenly along a highway. Each camera (SC) has a limited viewing range and cameras are placed to get an optimal coverage of the highway. Each camera has a communication unit to interact with other cameras. The goal of the cameras is to detect and monitor traffic jams on the highway in a decentralised way. The data observed by the multiple cameras have to be aggregated, so each camera has an agent that can play different roles in organizations. Agents exploit a distributed middleware, which provides support for dynamic organizations. This middleware acts as an AM; it encapsulates the management of dynamic evolution of organizations offering different roles to agents, based on the current context.

Pattern: *Knowledge-equipped Component*

– **Name:** *Knowledge-equipped Component*
– **Classification:** architecture, component, edlc-design, edlc-modeling
– **Intent:** Enable an autonomous component to operate in a context-sensitive manner that potentially requires interaction with other components.
– **Motivation:** Various architectures exist that allow components and systems to exhibit these kinds of complex, context-sensitive behaviors and interactions. *Knowledge-equipped Component*s are components with individual behaviors and knowledge repositories that can dynamically form aggregations.

These components can often be arranged in a *Flat Architecture* to provide a powerful and flexible, yet simple, architectural choice.

- **Context:** *Knowledge-equipped Component*s are well-suited to ensembles in which components need to act autonomously and interact with each other. They can be used in different architectural styles such as *Peer-to-peer* or *Client/Server* systems.

  Components need to have at least a modest amount of computational power and local storage; the pattern is not applicable for systems that rely on, e.g., pure stigmergy. Furthermore, if interaction is necessary, components must be equipped with a communication mechanism that enables sender and receiver to establish their identities and sufficient bandwidth must be available.
- **Description:** A knowledge-equipped component, is equipped with *behaviors* and a *knowledge repository.* Behaviors describe the computations each component performs. They are typically modeled as processes executing actions, for example in the style of process calculi or in rewriting logic. In systems using knowledge-equipped components, interaction between components is achieved by allowing components to access the knowledge repositories of other components; access restrictions are mediated by access policies.

  Knowledge repositories provide high-level primitives to manage pieces of information coming from different sources. Knowledge is represented through items containing either application data or awareness data. The former are used for determining the progress of component computations, while the latter provide information about the environment in which the components are running (e.g. monitored data from sensors) or about the actual status of an autonomic component (e.g. its current location). This allows components to be both context- and self-aware. The knowledge repository's handling mechanism for knowledge-equipped components has to provide at least operations for adding knowledge, as well as retrieving and withdrawing knowledge from it.
- **Implementation:** SCEL (see Chapter 1.2.1) defines primitives for modeling and implementing *Knowledge-equipped Component*s. An example for the behavior of a component implemented in SCEL is the following monitor for a garbage-collecting robot (which is a simplified version of the controller analyzed in [60]):

$$s \triangleq \mathbf{get}(item)@ctl.p$$
$$p \triangleq \mathbf{get}(items, !x)@master.\mathbf{put}(items, x+1)@master.c$$
$$c \triangleq \mathbf{get}(arrived)@ctl.\mathbf{put}(dropped)@master.s + \mathbf{get}(done)@ctl$$

  This monitor waits until a tuple *item* becomes available in the knowledge repository *ctl*, updates a counter in the knowledge repository *master*, and then waits until either a tuple *arrived* or a tuple *done* is available in *ctl*. In the first case the controller informs the repository *master* that it has *dropped* an item and resumes from the beginning, if instead a tuple *done* is retrieved from *ctl* the monitor stops.

This example also shows how several knowledge-equipped components can interact via a shared knowledge repository *master*. Note that no further synchronization primitives are necessary, even in the case where the *master* repository is shared between different components, since the first component to perform the action **get**$(items, !x)@master$ removes the *items*-tuple from this knowledge repository, and other components will block on their **get**$(items, !x)@master$ operations until the first component has **put** the updated tuple back into *master*.

- **Consequences:** A knowledge-equipped component can exhibit complex behavior that relies on local or shared knowledge. It can adapt its behavior flexibly to knowledge gathered while the ensemble is running. In some cases (e.g., some swarm robotics scenarios with limited sensor input) it may not be possible to extract the required knowledge from the available information. In general knowledge-equipped components have relatively high processing and storage requirements; shared knowledge repositories often require high network bandwidth and low latency.
- **Related Patterns:** The coordination of interactions for knowledge-equipped components is an example of *Tuple-space Based Coordination*; the interaction between components can be performed using *Attribute-based Communication*. If the knowledge of the component is repeatedly or continuously updated to correspond to the environment, the knowledge repository and processes responsible for updating it form an *Awareness Mechanism*. An ensemble containing multiple such components exhibits *Distributed Awareness-based Behavior*.

Pattern: *Statistical Model Checking*

- **Name:** *Statistical Model Checking*
- **Classification:** ensemble, validation, edlc-verification-and-validation
- **Intent:** Validate quantitative properties of a system at design time.
- **Motivation:** It is desirable to ascertain that a system can perform according to specification as early as possible in the design process, and to validate changes of the system design when requirements or environmental conditions change. Traditional verification and validation techniques are often difficult to scale to the size of ensembles.
- **Context:** *Statistical Model Checking* is applicable in many situations in which quantitative properties of ensembles need to be validated at design time. It is necessary to have (stochastic) models of the system and its environment that match the actual behavior closely enough to ensure meaningful results.
  While it scales well when compared to many other validation techniques, the computational and memory requirements of statistical model checking may be too high for very large systems. Systems that include non-determinism may pose problems for statistical model checkers, although advances in the area of statistical model checking for, e.g., Markov Decision Procedures, have recently been made. Statistical model checking provides only statistical

assurances; it can therefore not be applied in situations where a proof of correctness is required. Furthermore, statistical model checking cannot validate properties that can only be established for infinite execution traces. In cases where precise behavioral estimates are required, the effort for statistical model checking may be prohibitive.
- **Description:** In contrast to traditional (numerical) model checking techniques, statistical model checking runs simulations of the system, performs hypothesis testing on these simulations and then uses statistical estimates to determine whether the probability that the system satisfies the given hypotheses is above a certain threshold.
- **Applications:** Several examples for applying the *Statistical Model Checking* pattern to validate properties of ensembles and choose between different implementation strategies are presented in [29].

5 Related Work

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 [30] 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 [3] 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 [44] 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. [15] 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. [17] 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.

After the original "Gang of Four" book [35] introduced design patterns for object-oriented software development, pattern catalogues in various formats have been proposed for a large and varied number of domains, and covering many areas also addressed in the ASCENS pattern catalogue, e.g., application architecture [34, 33], distributed computing [25, 57, 5], testing [12], resource-constrained

devices [36], cooperative interactions [50], or fault-tolerant software [38]. However, many of the specific features of ASCENS and the EDLC, e.g., the use of formal methods or the integration of the design-time and runtime cycle are not addressed in these pattern languages.

In the last years the interest in engineering self-adaptive and autonomic systems is growing, as shown by the number of recent surveys and overviews on the topic [27, 56]. However, a comprehensive and rationally-organized analysis of architectural patterns for self-adaptation is still missing.

Salehie and Tahvildari [56] survey and classify the various principles underlying self-adaptation and the means by which adaptation ca be enforced in a system, i.e., the different mechanisms to promote adaptation at the behavioral and structural level. Similarly, Andersson et al. [4] propose a classification of modeling dimensions for self-adaptive systems to provide the engineers with a common set of vocabulary for specifying the self-adaptation properties under consideration and select suitable solutions. However, and although both these works emphasize the importance of feedback loops, nothing is said about the patterns by which such feedback loops can be organized to promote self-adaptation.

Coming to work that have a more direct relation with ours, Brun et al. [14] present a possible classification of self-adaptive systems with the emphasis on the use of feedback loops as first-class entities in control engineering. They unfold the role of feedback loops as a general mechanism for self-adaptation, essential for understanding all types of self-adaptation. Taking inspiration for control engineering, natural systems and software engineering, the authors present some self-adaptive architectures that exhibit feedback loops. They also identify the critical challenges that must be addressed to enable systematic and well-organized engineering of self-adaptive and self-managing software systems. Their analysis of different kinds of feedback loops is very relevant for our work, and in our effort towards a comprehensive and complete taxonomy of patterns for feedback loops we have partially built upon it.

Grounded on earlier works on architectural self-adaptation approaches, the FORMS model [59] (FOrmal Reference Model for Self-adaptation) enables engineers to describe, study and evaluate alternative design choices for self-adaptive systems. FORMS defines a shared vocabulary of adaptive primitives that – while simple and concise – can be used to precisely define arbitrary complex self-adaptive systems, and can support engineers in expressing their design choices, there included those related to the architectural patterns for feedback loops. FORMS does not have the ambition to analyse and classify architectural self-adaptation patterns, and rather has to be considered as a potentially useful complement to our work.

6 Conclusions

In this work we presented a software development life cycle for collective autonomic systems. Its aim is to support developers dealing with self-* properties of

ensembles, mainly self-awareness and self-adaptation talking into account environmental situations. A distinguishing feature of the double-wheeled life cycle is the feedback loop from runtime to design (in addition to the feedback loops at design and 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 behavior; indeed this provides a feedback loop that allows for improvement of an evolving software. We illustrated how the life cycle can be instantiated using a set of languages, methods and tools developed within the ASCENS project.

A first proof of concept of the life cycle was performed for the e-mobility domain [24]. Future plans are the validation of our engineering approach with more challenging scenarios of different application domains. A vision on future engineering approaches should consider to have a look at other disciplines even those that are not so directly related to computer science for ideas and technologies for building collective autonomic systems.

In addition, we have presented a catalog of patterns to provide reusable solutions for the development of collective autonomic systems. We included in the catalog patterns at different levels of abstraction so that the pattern catalog can also serve as introduction to certain development techniques. Therefore, it is useful for developers who are not already experts in the EDLC and the various technologies developed within the scope of the ASCENS project.

References

1. Abeywickrama, D.B., Zambonelli, F.: Model Checking Goal-Oriented Requirements for Self-Adaptive Systems. In: Proceedings of the 19th IEEE International Conference and Workshops on Engineering of Computer-Based Systems. pp. 33–42 (Apr 2012)
2. Abeywickrama, D., Bicocchi, N., Zambonelli, F.: Sota: Towards a general model for self-adaptive systems. In: Enabling Technologies: Infrastructure for Collaborative Enterprises (WETICE), 2012 IEEE 21st International Workshop on. pp. 48–53 (June 2012)
3. de Lemos et al., R.: Software Engineering for Self-Adaptive Systems: A second Research Roadmap. In: de Lemos, R., Giese, H., Müller, H., Shaw, M. (eds.) Software Engineering for Self-Adaptive Systems. No. 10431 in Dagstuhl Seminar Proceedings, Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, Germany, Dagstuhl, Germany (2011)
4. Andersson, J., et al.: Modeling dimensions of self-adaptive software systems. In: Cheng, B., et al. (eds.) Software Engineering for Self-Adaptive Systems, Lecture Notes in Computer Science, vol. 5525, pp. 27–47. Springer (2009)
5. Babaoglu, O., Canright, G., Deutsch, A., Caro, G.A.D., Ducatelle, F., Gambardella, L.M., Ganguly, N., Jelasity, M., Montemanni, R., Montresor, A., Urnes, T.: Design patterns from biology for distributed computing. ACM Trans. Auton. Adapt. Syst. 1(1), 26–66 (2006)
6. Basu, A., Bensalem, S., Bozga, M., Caillaud, B., Delahaye, B., Legay, A.: Statistical abstraction and model-checking of large heterogeneous systems. In: Formal

Techniques for Distributed Systems, Joint 12th IFIP WG 6.1 International Conference, FMOODS 2010 and 30th IFIP WG 6.1 International Conference, FORTE 2010, Amsterdam, The Netherlands, June 7-9, 2010. Proceedings. pp. 32–46 (2010)
7. Basu, A., Bensalem, S., Bozga, M., Combaz, J., Jaber, M., Nguyen, T.H., Sifakis, J.: Rigorous component-based design using the BIP framework. IEEE Software, Special Edition – Software Components beyond Programming – from Routines to Services 28(3), 41–48 (2011)
8. Basu, A., Bozga, M., Sifakis, J.: Modeling Heterogeneous Real-time Components in BIP. In: SEFM. pp. 3–12. IEEE Computer Society (2006)
9. Bensalem, S., Bozga, M., Delahaye, B., Jégourel, C., Legay, A., Nouri, A.: Statistical Model Checking QoS Properties of Systems with SBIP. In: Margaria, T., Steffen, B. (eds.) ISoLA (1). LNCS, vol. 7609, pp. 327–341. Springer (2012)
10. Bensalem, S., Bozga, M., Nguyen, T.H., Sifakis, J.: D-Finder: A Tool for Compositional Deadlock Detection and Verification. In: Bouajjani, A., Maler, O. (eds.) CAV. LNCS, vol. 5643, pp. 614–619. Springer (2009)
11. Bensalem, S., Griesmayer, A., Legay, A., Nguyen, T.H., Peled, D.: Efficient Deadlock Detection for Concurrent Systems. In: Singh, S., Jobstmann, B., Kishinevsky, M., Brandt, J. (eds.) MEMOCODE. pp. 119–129. IEEE (2011)
12. Binder, R.: Testing Object-Oriented Systems: Models, Patterns and Tools. Addison-Wesley Professional (2000)
13. Bröckers, A., Lott, C.M., Rombach, H.D., Verlage, M.: MVP-L Language Report Version 2. Tech. Rep. Technical Report Nr. 265/95, University of Kaiserslautern (1995)
14. Brun, Y., et al.: Engineering self-adaptive systems through feedback loops. In: Cheng, B., et al. (eds.) Software Engineering for Self-Adaptive Systems, Lecture Notes in Computer Science, vol. 5525, pp. 48–70. Springer (2009)
15. Brun, Y., Marzo Serugendo, G., Gacek, C., Giese, H., Kienle, H., Litoiu, M., Müller, H., Pezzè, M., Shaw, M.: Software engineering for self-adaptive systems. chap. Engineering Self-Adaptive Systems through Feedback Loops, pp. 48–70. Springer, Berlin, Heidelberg (2009)
16. Bruni, R., Corradini, A., Gadducci, F., Hölzl, M., Lafuente, A.L., Vandin, A., Wirsing, M.: Reconciling White-Box and Black-Box Perspectives on Behavioural Self-Adaptation. In: Wirsing, M., Hölzl, M., Koch, N., Mayer, P. (eds.) Software Engineering for Collective Autonomic Systems: Results of the ASCENS Project, Lecture Notes in Computer Science, vol. 8998. Springer Verlag, Heidelberg (2015)
17. Bruni, R., Corradini, A., Gadducci, F., Lluch-Lafuente, A., Vandin, A.: A Conceptual Framework for Adaptation. In: de Lara, J., Zisman, A. (eds.) FASE. LNCS, vol. 7212, pp. 240–254. Springer (2012)
18. Bulej, L., Bures, T., Gerostathopoulos, I., Horky, V., Keznikl, J., Marek, L., Tschaikowski, M., Tribastone, M., Tuma, P.: Supporting Performance Awareness in Autonomous Ensembles. In: Wirsing, M., Hölzl, M., Koch, N., Mayer, P. (eds.) Software Engineering for Collective Autonomic Systems: Results of the ASCENS Project, Lecture Notes in Computer Science, vol. 8998. Springer Verlag, Heidelberg (2015)
19. Bulej, L., Bures, T., Horky, V., Keznikl, J., Tuma, P.: Performance Awareness in Component Systems: Vision Paper. In: Proceedings of COMPSAC 2012. COMPSAC '12 (2012)
20. Bulej, L., Bures, T., Keznikl, J., Koubkova, A., Podzimek, A., Tuma, P.: Capturing Performance Assumptions using Stochastic Performance Logic. In: Proc. 3rd Intl. Conf. on Performance Engineering. ICPE'12, Boston, MA, USA (2012)

21. Bulej, L., Bure, T., Keznikl, J., Koubkov, A., Podzimek, A., Tma, P.: Capturing performance assumptions using stochastic performance logic. In: Proc. ICPE 2012. pp. 311–322. ICPE 2012, ACM, New York, NY, USA (2012)
22. Bures, T., Gerostathopoulos, I., Hnetynka, P., Keznikl, J., Kit, M., Plasil, F.: DEECO: An Ensemble-Based Component System. In: Proceedings of the 16th International ACM Sigsoft symposium on Component-based software engineering. pp. 81–90. CBSE '13, ACM, New York, NY, USA (2013)
23. Bures, T., Gerostathopoulos, I., Hnetynka, P., Keznikl, J., Kit, M., Plasil, F.: The Invariant Refinement Method. In: Wirsing, M., Hölzl, M., Koch, N., Mayer, P. (eds.) Software Engineering for Collective Autonomic Systems: Results of the ASCENS Project, Lecture Notes in Computer Science, vol. 8998. Springer Verlag, Heidelberg (2015)
24. Bures, T., Nicola, R.D., Gerostathopoulos, I., Hoch, N., Kit, M., Koch, N., Monreale, G.V., Montanari, U., Pugliese, R., Serbedzija, N., Wirsing, M., Zambonelli, F.: A life cycle for the development of autonomic systems: The e-mobility showcase. 2013 IEEE 7th International Conference on Self-Adaptation and Self-Organizing Systems Workshops 0, 71–76 (2013)
25. Buschmann, F., Henney, K., Schmidt, D.C.: A Pattern Language for Distributed Computing, Pattern-Oriented Software Architecture, vol. 4. Wiley (2007)
26. Cabri, G., Puviani, M., Zambonelli, F.: Towards a Taxonomy of Adaptive Agent-Based Collaboration Patterns for Autonomic Service Ensembles. In: Proceedings of the 2011 International Conference on Collaboration Technologies and Systems. pp. 508–515. IEEE (May 2011)
27. Cheng, B., et al.: Software engineering for self-adaptive systems: A research roadmap. In: Cheng, B., et al. (eds.) Software Engineering for Self-Adaptive Systems, Lecture Notes in Computer Science, vol. 5525, pp. 1–26. Springer (2009)
28. Combaz, J., Bensalem, S., Kofron, J.: Correctness of Service Components and Service Component Ensembles. In: Wirsing, M., Hölzl, M., Koch, N., Mayer, P. (eds.) Software Engineering for Collective Autonomic Systems: Results of the ASCENS Project, Lecture Notes in Computer Science, vol. 8998. Springer Verlag, Heidelberg (2015)
29. Combaz, J., Lafuente, A.L., Montanari, U., Pugliese, R., Sammartino, M., Tiezzi, F., Vandin, A., von Essen, C.: Verification Results Applied to the Case Studies - ASCENS Joint Deliverable JD3.1 (2013), http://www.ascens-ist.eu/deliverables/JD31.pdf
30. Corporation, I.: An Architectural Blueprint for Autonomic Computing. Tech. rep., IBM (2005), http://researchr.org/publication/autonomic-architecture-2005
31. De Nicola, R., Ferrari, G.L., Loreti, M., Pugliese, R.: A Language-Based Approach to Autonomic Computing. In: Formal Methods for Components and Objects, 10th International Symposium, Revised Selected Papers. pp. 25–48 (2011)
32. De Nicola, R., Loreti, M., Pugliese, R., Tiezzi, F.: SCEL: A Language for Autonomic Computing. Tech. rep., IMT Lucca (January 2013)
33. Erl, T.: SOA Design Patterns. Prentice Hall PTR, Upper Saddle River, NJ, USA, 1st edn. (2009)
34. Fowler, M.: Patterns of Enterprise Application Architecture. Addison-Wesley Longman, Amsterdam (November 2002)
35. Gamma, E., Helm, R., Johnson, R., Vlissides, J.: Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley Longman, Amsterdam (1995)
36. Gay, D., Levis, P., Culler, D.: Software design patterns for TinyOS. Trans. on Embedded Computing Sys. 6(4), 22 (2007)

37. Gomaa, H., Hashimoto, K.: Dynamic self-adaptation for distributed service-oriented transactions. In: International Workshop on Software Engineering for Adaptive and Self-Managing Systems. pp. 11–20. IEEE, Zurich, Switzerland (2012)
38. Hanmer, R.: Patterns for Fault Tolerant Software. John Wiley & Sons, 1 edn. (Oct 2007)
39. Hölzl, M.: APEX: The ASCENS Pattern Explorer. web site, http://www.ascens-ist.eu/pattern
40. Hölzl, M.: The Poem Language (Version 2). Tech. Rep. 7, ASCENS (July 2013), http://www.poem-lang.de/documentation/TR7.pdf
41. Hölzl, M., Gabor, T.: Reasoning and Learning for Awareness and Adaptation. In: Wirsing, M., Hölzl, M., Koch, N., Mayer, P. (eds.) Software Engineering for Collective Autonomic Systems: Results of the ASCENS Project, Lecture Notes in Computer Science, vol. 8998. Springer Verlag, Heidelberg (2015)
42. Hölzl, M., Koch, N.: D8.3: Third Report on WP8—Best Practices for SDEs (first version) (November 2013)
43. Hölzl, M.M., Wirsing, M.: Towards a system model for ensembles. In: Agha, G., Danvy, O., Meseguer, J. (eds.) Formal Modeling: Actors, Open Systems, Biological Systems. LNCS, vol. 7000, pp. 241–261. Springer (2011)
44. Inverardi, P., Mori, M.: A Software Lifecycle Process to Support Consistent Evolutions. In: de Lemos, R., Giese, H., Müller, H.A., Shaw, M. (eds.) Software Engineering for Self-Adaptive Systems. LNCS, vol. 7475, pp. 239–264. Springer (2010)
45. Kephart, J.O., Chess, D.M.: The vision of autonomic computing. Computer 36(1), 41–50 (Jan 2003), http://dx.doi.org/10.1109/MC.2003.1160055
46. Keznikl, J., Bures, T., Plasil, F., Gerostathopoulos, I., Hnetynka, P., Hoch, N.: Design of Ensemble-Based Component Systems by Invariant Refinement. In: Proceedings of the 16th International ACM Sigsoft symposium on Component-based software engineering. pp. 91–100. CBSE '13, ACM, New York, NY, USA (2013)
47. Keznikl, J., Bures, T., Plasil, F., Kit, M.: Towards Dependable Emergent Ensembles of Components: The DEECo Component Model. In: WICSA/ECSA. pp. 249–252. IEEE (2012)
48. Klarl, A., Hennicker, R.: Design and Implementation of Dynamically Evolving Ensembles with the HELENA Framework. In: Proceedings of the 23rd Australasian Software Engineering Conference. pp. 15–24. IEEE (2014)
49. Marek, L., Villazón, A., Zheng, Y., Ansaloni, D., Binder, W., Qi, Z.: DiSL: a domain-specific language for bytecode instrumentation. In: Proceedings of the 11th annual international conference on Aspect-oriented Software Development. pp. 239–250. AOSD '12, ACM, New York, NY, USA (2012)
50. Martin, D., Sommerville, I.: Patterns of Cooperative Interaction: Linking Ethnomethodology and Design. ACM Trans. Comput.-Hum. Interact. 11(1), 59–89 (2004)
51. Morandini, M., et al.: On the use of the goal-oriented paradigm for system design and law compliance reasoning. In: iStar 2010–4 th International i* Workshop. p. 71. Hammamet, Tunisia (2010)
52. Mylopoulos, J., Chung, L., Yu, E.S.K.: From Object-Oriented to Goal-Oriented Requirements Analysis. Communications of the ACM 42(1), 31–37 (1999)
53. Nicola, R.D., Latella, D., Lafuente, A.L., Loreti, M., Margheri, A., Massink, M., Morichetta, A., Pugliese, R., Tiezzi, F., Vandin, A.: The SCEL Language: Design, Implementation, Verification. In: Wirsing, M., Hölzl, M., Koch, N., Mayer, P. (eds.) Software Engineering for Collective Autonomic Systems: Results of the ASCENS Project, Lecture Notes in Computer Science, vol. 8998. Springer Verlag, Heidelberg (2015)

54. Owre, S., Rushby, J.M., Shankar, N.: PVS: A prototype verification system. In: Kapur, D. (ed.) Automated Deduction - CADE-11, 11th International Conference on Automated Deduction, Saratoga Springs, NY, USA, June 15-18, 1992, Proceedings. Lecture Notes in Computer Science, vol. 607, pp. 748–752. Springer (1992), http://dx.doi.org/10.1007/3-540-55602-8_217
55. Puviani, M.: Catalogue of architectural adaptation patterns (2012), http://mars.ing.unimo.it/wiki/papers/TR42.pdf
56. Salehie, M., Tahvildari, L.: Self-adaptive software: Landscape and research challenges. ACM Transactions on Autonomous and Adaptive Systems 4(2), 1–42 (2009)
57. Schmidt, D.C., Stal, M., Rohnert, H., Buschmann, F.: Patterns for Concurrent and Networked Objects, Pattern-Oriented Software Architecture, vol. 2. Wiley (2000)
58. Vassev, E., Hinchey, M.: Engineering Requirements for Autonomy Features. In: Wirsing, M., Hölzl, M., Koch, N., Mayer, P. (eds.) Software Engineering for Collective Autonomic Systems: Results of the ASCENS Project, Lecture Notes in Computer Science, vol. 8998. Springer Verlag, Heidelberg (2015)
59. Weyns, D., Malek, S., Andersson, J.: Forms: Unifying reference model for formal specification of distributed self-adaptive systems. ACM Transactions on Autonomous and Adaptive Systems 7(1), 8 (2012)
60. Wirsing, M., Hölzl, M.M., Tribastone, M., Zambonelli, F.: ASCENS: Engineering Autonomic Service-Component Ensembles. In: Beckert, B., Damiani, F., de Boer, F.S., Bonsangue, M.M. (eds.) FMCO. Lecture Notes in Computer Science, vol. 7542, pp. 1–24. Springer (2011)