www.ascens-ist.eu

ASCENS

Autonomic Service-Component Ensembles

D8.3: Third Report on WP8

Best Practices for SCEs (first version)

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): **Matthias Hölzl, Nora Koch (LMU)**

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

This deliverable reports on the activities of work package 8 in the third reporting period. The main activities were the development of the Ensemble Development Life Cycle (EDLC) and the preparation of a catalogue of patterns describing best practices for ensemble engineering.

The EDLC provides a framework that puts the work done in the other work packages *"into a larger perspective from an engineering point of view, thereby identifying best practices for service component ensembles."* An in-depth overview of the EDLC is given in joint deliverable 3.2 [KBC+13], therefore this deliverable gives only a short overview of the EDLC.

The catalogue of patterns codifies *"best practices discovered during the project in a form that is easily accessible for SCE practitioners."* In this deliverable we present the first version of the ASCENS pattern catalogue, discuss the kinds of patterns included in the catalogue as well as the pattern language used for writing the patterns, and we present some exemplary patterns from the catalogue.

Contents

1 Introduction

The Description of Work of the ASCENS project states as general objective for work package 8 that *"[...] the foundational work done of ASCENS, applied to the case studies in work package 7 must be put into a larger perspective from an engineering point of view, thereby identifying best practices for service component ensembles."* The two ongoing tasks of WP8 are T8.2 *"An SCs component repository for self-aware autonomic ensembles"* which was reported in deliverable D8.2 [HBGK13], and T8.3 *"Best Practices for SCEs."* The main objective of T8.3 is *"to codify best practices discovered during the project in a form that is easily accessible for SCE practitioners. This work will result in a catalogue of SCE patterns for the overall results of the ASCENS project."*

1.1 Results Obtained During the Work Period

The two main results for Task T8.3 in this reporting period are the definition of the Ensemble Development Life Cycle (EDLC) and an initial version of the catalogue of patterns.

The EDLC provides the larger perspective mentioned in the general objectives for work package 8 and represents the majority of the work done in this work package during this reporting period. Details about the EDLC can be found in JD3.2 [KBC+13].

The second work item in task T8.3, a catalogue of patterns, will present best practices for the development of ensembles according to the EDLC and using the results of the project in a form that is easily accessible for practitioners. The pattern language used in the catalogue is strongly influenced by the adaptation patterns developed as part of work package 4 but it has been augmented to take into account the needs of the other work packages of ASCENS.

1.2 Connection to Other Work Packages

Since the EDLC ties together all research areas of the ASCENS project into a coherent whole, it was developed in cooperation with all technical work packages of the project.

The patterns in the catalogue are based on the work of most other work packages and informed by the application of this work to the case studies of WP7. Many patterns try to present aspects of the work performed in the technical work packages from a software engineering perspective; examples are patterns such as *Tuple-space Based Coordination*, *Knowledge-equipped Component* (WP1), *Soft-constraint-based Optimization* (WP2), *Build Small Ontology* (WP3), the awareness patterns (WP4), and *Statistical Model Checking* (WP5).

1.3 Overview of this Deliverable

In the next section we give a short overview of the EDLC; an in-depth discussion can be found in joint deliverable JD3.2 [KBC+13]. In the subsequent sections we focus on the catalogue of SCE patterns. Sect. 3 describes the pattern language used in the pattern catalogue, as well as a tool for interactive exploration of the pattern catalogue and the creation of new patterns. In Sect. 4 we show the definition of several patterns and how they might be used by a developer. The final section presents the planned work for the fourth reporting period and concludes.

2 Overview of the Ensemble Development Life Cycle

The development of adaptive and autonomic systems requires that certain activities that are usually performed by human developers before deploying the system, are deferred to run-time and autonomously performed by the system itself. Therefore the development process can no longer be

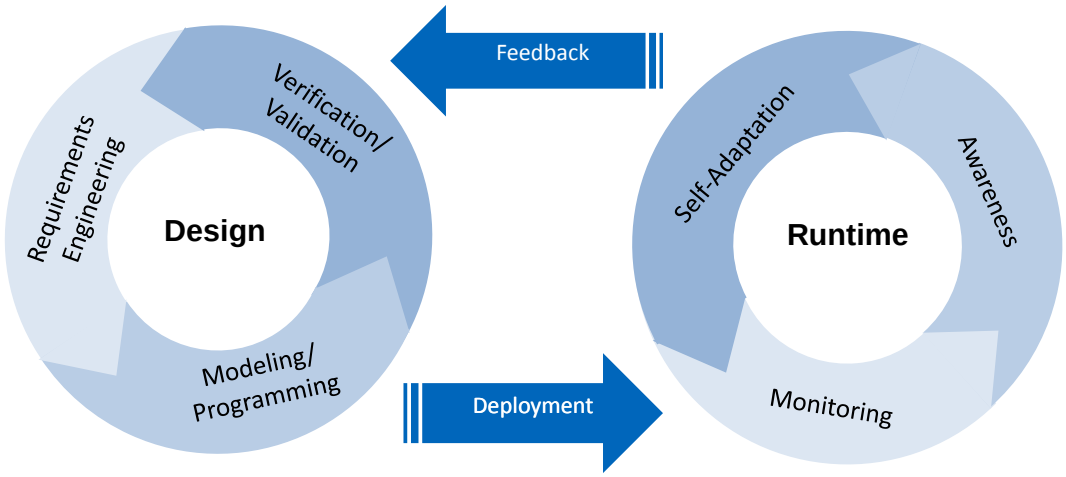

Figure 1: Ensemble Development Life Cycle (EDLC)

described as a single waterfall or spiral that details development activities undertaken at design time and produces as result a system that is ready for deployment; instead the deployment and operation of the system have to be entwined with the traditional design-time activities resulting in an ongoing interaction between development and runtime activities.

To capture this co-dependency between development and operations, we propose a "double-wheel" life cycle for the engineering process as shown in Figure 1. The "left wheel" represents the *design* or *offline* phases and the right one the *runtime* or *online* phases. Both wheels are connected by the *deployment* and *feedback* transitions.

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.

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 and possibly processed by the system's awareness mechanism are fed back to the design cycle to provide information to be used for system redesign, verification and redeployment.

A detailed description of the EDLC is given in JD3.2 [KBC+13]. In the rest of this deliverable we will focus on the catalogue of patterns that supports the development of service components and service-component ensembles.

3 The ASCENS Pattern Languages and Patterns

Pattern languages and patterns have been developed for a wide range of domains. After the original "Gang of Four" book [GHJV95] 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 for development problems ranging from implementation choices to system architecture.

While it is possible to describe a single design solution in the form of a pattern, it is common to present patterns in the form of a *pattern language* or *pattern system:* a catalog of patterns written

using the same format with cross-references between patterns. Typical pattern languages contain, for example, links between alternative patterns that represent different trade-offs for a design problem, links from structural patterns to patterns useful for implementing that structure, and so on.

Presenting design solutions in the form of a pattern language has several advantages:

- The standardized structure of patterns belonging to the same pattern language makes it easy for developers to determine which patterns fit their domain and the problem they are trying to solve.
- Typically patterns contain explicit information about the advantages *and disadvantages* of certain design choices, thereby alerting developers to costs or undesirable side effects for using a certain pattern.
- When patterns are cross-linked in a pattern language, connections, such as alternatives for different design problems and patterns that complement each other are clearly visible. In many cases patterns contain links to other patterns that might be applicable in situations where the original pattern is not a good solution.

3.1 A Pattern Language for Ensembles

The development of ensembles poses many interrelated design challenges and implementation choices; describing them via a pattern language makes it easier for developers to comprehend 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 catalogue can also serve as introduction to certain development techniques. Currently our pattern catalogue 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* (see Sect. 4.4) that describes the general concept of awareness mechanisms.

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* (see Sect. 4.2). These patterns often serve as entry points into the catalogue for developers trying to solve an architectural problem.

Adaptation Patterns: Patterns concerned with adaptation and the control-loop structure of ensembles, as reported in D4.2 [ZAC+12] and [Puv12]. An example for a pattern in this area is *Reactive Stigmergy Service Components Ensemble* (see [ZAC+12, p.21]).

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 don't 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 catalogue 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 downsides of using the anti-pattern.

When exploring the pattern catalogue, 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 catalogue.

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. For example, *Algorithmic Planning* is a generic pattern that is specialized by several other patterns with very different areas of applicability. In [KBC+13, Sect. 3.3.5] the *HTN Planning* specialization of *Algorithmic Planning* is used to improve the navigation of a robot; because of the resource constraints in this scenario most other variants of *Algorithmic Planning* would not be applicable.

Aliases: Other names by which this pattern is known.

Intent:* The purpose for this pattern, what does the pattern accomplish? For example, for *Algorithmic Planning* the intent is "Enable agents to dynamically satisfy goals or compose complex behaviors. Perform *Temporal Decomposition* of tasks."

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. The motivation given in the pattern for *Algorithmic Planning* is "In many environments it is feasible to determine the preconditions and effects that simple actions have, but it is difficult to pre-compute at design-time algorithms for all complex activities that a system might have to perform. Under these conditions, *Algorithmic Planning* can be used to compute behaviors that satisfy the system's goals based on descriptions of the simple actions that are possible."

Applicability:* Describes for which systems the pattern is applicable, and which influences might lead to other patterns being preferable. In the case of *Algorithmic Planning*, the applicability section starts as follows: "The system has to be able to determine relevant states of its environment and it has to possess enough computational power and storage to perform the planning process. In addition, most planning systems assume that the world is deterministic and that no contingencies arise during the execution of the plan. Planning methods such as *MDP-based Planning* can deal with uncertain worlds but typically place very high computational demands on the system. . . . "

Classification:* The set of keywords that describes, e.g., to which phases of the EDLC the pattern applies. For *Algorithmic Planning* the keywords are "awareness, component, edlc-awareness," i.e., it is an awareness pattern that is relevant to the "awareness" phase of the EDLC and it is mostly concerned with individual components.

Description/Behavior:* A description of the pattern. For *Algorithmic Planning* the description contains an overview of automated planning, references to different planning mechanisms and the restrictions imposed by them.

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 SOTA specification of their behavior.

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, e.g., the *Algorithmic Planning* pattern references the PIRLO system [Bel13] which supports action programming in rewriting logic as well as the Iliad runtime which supports HTN- and MDP-based planning, among others.

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. For *Algorithmic Planning* the patterns *HTN Planning* and *MDP-based Planning* appear in this section.

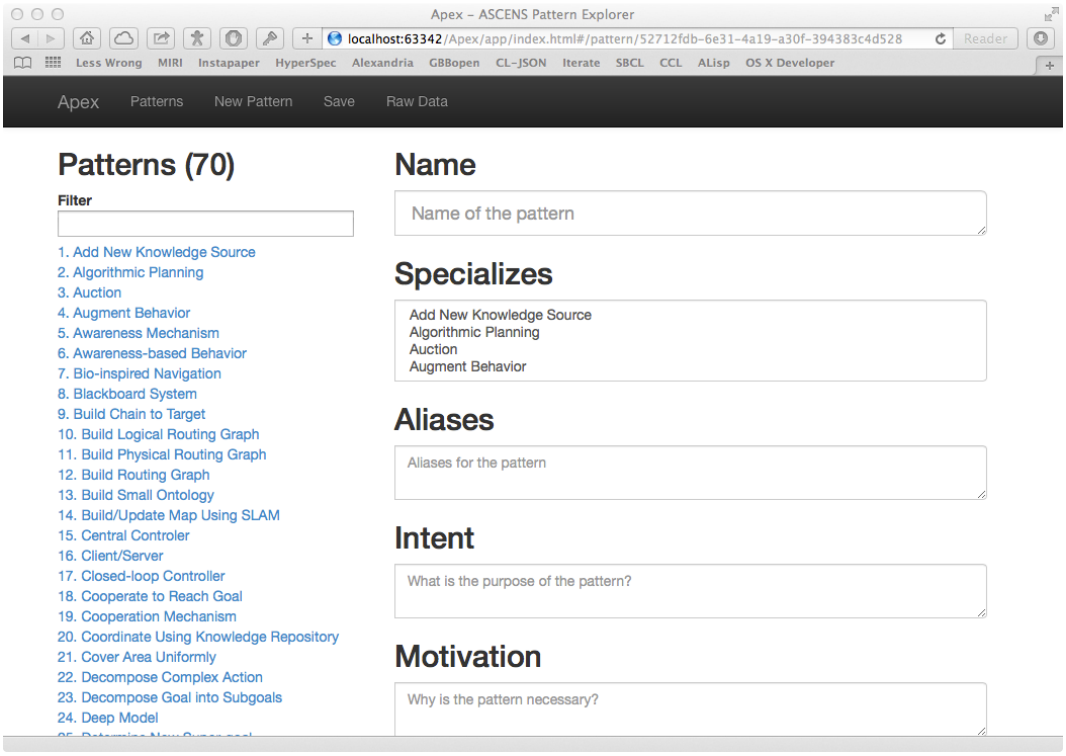

Figure 2: The Ascens Pattern Explorer (APEX): Creating a new pattern

Applications: References to applications in which this pattern was used.

This pattern language 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.

3.2 The ASCENS Pattern Explorer

The patterns in any pattern language form an intricate web of relationships that is as important as the contents of the individual pattern. To simplify the exploration of our pattern catalogue we have developed the ASCENS Pattern Explorer (APEX) which allows developers to view the list of patterns, or to filter this list using either a full-text search in the pattern description or values of specific fields. Occurrences of Patterns are cross-linked throughout the text of the pattern. This simplifies the search for patterns that match a particular problem, and it also facilitates the exploration of the pattern catalogue.

Users with sufficient permissions can also enter new patterns into APEX using a wiki-like interface. Since the sections of the pattern are provided by the application, new patterns can easily be entered without remembering details of the pattern template. A screenshot of the edit screen of APEX is shown in Fig. 2.

The APEX pattern catalogue currently contains 70 entries.[1] A first version of 24 of these patterns has been completed, work on the other patterns is in progress.

[1]This number does not yet include the adaptation patterns contained in [ZAC+12] since they have not yet been entered into APEX.

4 Selected Patterns

This section contains a selection of patterns from the ASCENS pattern catalogue. The names of all patterns are set in *Italics and Titlecase*; the patterns have been slightly edited to remove repetition that is necessary to provide context when patterns are read individually and to make them more suitable for presentation in a deliverable. The name field of the patterns is used as the section heading and not repeated in the text.

To illustrate the network structure of the pattern language and the potential to navigate from high-level pattern to low-level implementation advice (and *vice versa*), the patterns in this section are arranged in the way that a developer might traverse the links of the pattern catalogue. We assume that the developer starts by looking for an architecture of the component they are developing, and arrives at the pattern *Knowledge-equipped Component*. When reading the "Related Patterns" section, the developer might decide to investigate the effect this choice has on the ensemble by following the link to the pattern *Awareness-based Behavior*. The "Dynamic Behavior" section of this pattern mentions *Statistical Model Checking* as an appropriate validation technique, so the developer might follow this link, return to the *Awareness-based Behavior* pattern, and then investigate the link to *Awareness Mechanism*s. From there, the developer might continue to investigate links to suitable implementation techniques for awareness mechanisms, and arrive at patterns such as *MDP-based Planning*, *Data Driven Execution* or *Tuple-space Based Coordination* (not shown in this deliverable) that are relevant for the implementation of the component.

4.1 Pattern: *Knowledge-equipped Component*

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.

Applicability

*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.

Classification

architecture, component, edlc-design, edlc-modeling

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 [DLPT13] 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 [WHTZ11]):

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

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 $\mathbf{get}(items, !x)@master$ removes the *items*-tuple from this knowledge repository, and other components will block on their $\mathbf{get}(items, !x)@master$ operations until the first component has **put** the updated tuple back into *master*.

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*.

4.2 Pattern: *Distributed Awareness-based Behavior*

Intent

Enable an ensemble to operate in complex, open-ended, partially observable environments by providing individual components with *Awareness Mechanism*s.

Motivation

Ensembles often have to operate in environments whose properties are not completely known; they may encounter situations where complex decisions have to be taken under uncertain circumstances. Predetermining the desired behaviors at design time is often difficult or impossible. Furthermore, often individual components of an ensemble have to act autonomously. In these cases it may be possible to achieve the desired behavior of the ensemble by equipping individual components with an *Awareness Mechanism* and provide them with goals or measures of utility so that the aggregate behavior of the components will result in the desired system behavior.

Applicability

For this pattern to be applicable, structural information about the environment has to be available as an awareness model (see *Awareness Mechanism*); this model may be a database-like mechanism such as a tuple space, or it may take the more complex form of a *Logical Model* or a *Probabilistic Model* with corresponding reasoners. The components have to be able to map their runtime situation to the awareness model with enough precision to ensure that the quality of the awareness mechanism is high enough to reliably reach the system's goals. The overall task of the system has to be reducible to tasks that can (possibly cooperatively) be performed by the individual components; the distribution of these subtasks to the components has to be decided at design time or a run-time mechanism for *Task Allocation* has to be feasible. For a utility-based system it must be possible to factor the system's overall utility function into individual utility functions for each component (i.e., perform *Threadwise Decomposition* of the utility function). Depending on the size and complexity of the awareness model, significant computational resources and storage might have to be available at run time.

When these requirements are not satisfied, *Model-free Learning* applied to individual components may provide some of the same benefits but potentially require a large number of training runs and offer fewer static guarantees. If the system's run-time resources are not sufficient for providing an awareness model but data about the circumstances encountered by the system can be transmitted back to the developers, *Simulation in the Loop* may be used to reduce the amount of work that has to be performed by the awareness mechanism of the system. If at least some components can be equipped with sufficient capabilities to run awareness mechanisms, a *Teacher/Student* architecture can be employed to reduce the computational effort on a subset of components. If all components have severe resource restrictions, an awareness-based architecture may not be possible and system architectures such as *Swarm* or *Client/Server* may be more appropriate.

Classification

awareness, architecture, edlc-design, ensemble

Description

The architecture of a system based on this pattern is relatively straightforward, but each of the following steps poses significant challenges and is detailed by further patterns. In a system based on

Distributed Awareness-based Behavior, each component is equipped with an *Awareness Mechanism*. The tasks each component has to perform are either determined at design time, or a method to perform *Task Allocation* or *Threadwise Decomposition* of the system's utility function is used at run time to divide the system tasks among the system's components. Further *Temporal Decomposition* of the component's goals or utility functions is often necessary to reduce goals to atomic operations. Unless a decomposition can be found that allows each decomposed task to be performed by a single component, a *Cooperation Mechanism* has to be integrated into the system.

Dynamic Behavior

The characteristic feature of an architecture based on *Distributed Awareness-based Behavior* is that each component has its own awareness mechanism; therefore no general description of the ensemble's behavior or the dynamic behavior of components in an ensemble using this architecture is possible.

A simple control loop for components with an awareness mechanism might take the following form:

1. Determine the next task to be performed
2. Decompose the task into executable subtasks
3. Perform the subtasks

In this case the bulk of the work in the first two steps would be performed by the awareness mechanism. In practice, however, components are often structured as *Closed Loop Controller*s so that the performance of the subtasks can be monitored and corrective action taken if the results of the actual execution do not match the predicted situation.

Ensuring that the dynamic behavior of systems bases on autonomous components matches their specification is a difficult problem. In many cases, *Design-time Verification* and *Design-time Validation* techniques can be used. In particular, *Statistical Model Checking* techniques can often be applied fruitfully.

Implementation

In the ASCENS project, the jRESP [BHK+12] implementation of SCEL [DLPT13] is typically used to define the overall behavior of the ensemble, the communication and coordination between components and the overall behavior of individual components. The awareness mechanism of each component performs reasoning tasks for the SCEL controller of this component; it can be implemented, e.g., in *KnowLang* [Vas12a] using the *KnowLang* reasoner or in POEM [Höl13] using the Iliad/jRESP integration.

An example for an awareness mechanism for a single component that forms a *Closed Loop Controller* is given in the description of the *Awareness Mechanism* pattern, pp. 19*ff*.

4.3 Pattern: *Statistical Model Checking*

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.

Applicability

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.

Classification

component, edlc-verification-and-validation, ensemble, validation

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.

Since it samples the execution traces, statistical model checking has several advantages over numerical model checking [LDB10]:

- Only the distribution of sample executions has to be represented in the model, and it is possible to use, e.g., infinite state models or "black-box" models in which part of the internal structure is unknown or not yet decided.
- Since the model-checking process works by hypothesis testing, a wider range of logics can be used to describe the desired system behavior than for other model-checking approaches.
- It is easy to parallelize the model checking process, since several simulations can be run concurrently.

On the other hand, the results obtained by statistical model checking are only statistical estimates and the computational effort to obtain small confidence intervals may be very high.

Examples

Several examples for applying the *Statistical Model Checking* pattern to validate properties of ensembles and choose between different implementation strategies are presented in [Be13].

4.4 Pattern: Awareness Mechanism

Intent

An awareness mechanism is a model of an ensemble's environment (and possibly the ensemble itself) that is available at run-time, together with reasoners, and a sensor system that keeps the model in sync with the environment. Awareness mechanisms can be used to reason about the current state of the environment and provide an *Illusion of Stability* for deterministic reasoners performing, e.g., *Algorithmic Planning* or *Action-calculus Reasoning*.

Summary

An *awareness mechanism* consists of an *awareness model* which is *inversely connected* to the environment, *reasoners* that can draw inferences from the awareness model, and a *sensor system* that maintains the inverse connection. See below for a description of these terms and [HW14] for a detailed discussion. An awareness model is often a *Multi-Model* and a *Deep Model*, in particular it often combines *Logical Model*s and *Probabilistic Model*s.

Classification

awareness, conceptual

Description

We introduce the elements that an awareness mechanism has to possess, and the characteristics according to which awareness mechanism can be classified.

Elements of the awareness mechanism It is difficult to conceive of a completely memory-less (self-)aware system; therefore a system S that exhibits awareness has to have *some* way to store information about itself or its environment E; we call this information the *awareness model* M of S. The awareness model can be distributed among various nodes of S and even the environment E. Therefore, our concept of awareness models also encompasses system architectures based on stigmergy, e.g., robots that place tokens in their environment to mark places they have already visited.

The designers of a system that operates in a well-known, static environment may build an internal model that contains all information required by the system to operate successfully. In the more interesting case of open-ended, non-deterministic environments in which other agents are operating as well, we cannot ignore the dynamics of the environment E (which includes, from the point of view of S, the other agents): we want changes in E to influence the awareness model M. However this influence is usually not immediate, since the system has to obtain the information that E has changed before it can update M. Therefore we say that S (or, slightly inaccurately, M) is *inversely (causally) connected* to E if certain changes in E lead to corresponding changes in M after S reaches some state in which it can perceive the changes in its environment. We call the subsystem that is responsible for maintaining the inverse connection between E and M the *sensor system* of S. Apart from sensors or other data input devices it may contain pre-processing or filtering units that transform the raw data in a form that is more amenable to future processing.

Most environments are only partially observable: S cannot directly perceive all relevant information, instead it may have to *reason* about the available data to obtain the information required for action. We use the term "reason" in a very broad sense: the *reasoning engine* of a simple agent might be a program that simply queries the data stored in its awareness model, or it might perform simple computations, such as computing the length of a path by summing up the length of its components.

More sophisticated reasoning engines might perform complex inferences, run simulations or develop plans as part of their reasoning process, and a system may include several, distributed reasoning engines.

We call the combination of sensor system, awareness model and reasoning engines of a system its *awareness mechanism.* Its components need not be dedicated to the awareness mechanism, they can also be used by other parts of the system.

A system's awareness mechanism provides no benefits unless other parts of S or external observers have some way to access its contents. In the simplest case, access may be provided an interface that allows other components of the system to pose queries to the reasoners and retrieve the results of the inference; in biological systems this mechanism may be much more complicated and directly integrated with the awareness model and reasoners.

If it allows queries about the history of the system we say that the access mechanism is *diachronic*; if S can query its awareness mechanism about consequences of actions without committing to these actions or about properties of hypothetical environments we say it *allows hypotheses*.

A white-box definition of awareness The presence or absence of a certain class of model does not separate ensembles into "aware" or "not aware;" instead we want to characterize and compare the *degree of awareness* that various ensembles possess. In this section we focus on a white-box definition in which we assume that we can analyze all internal mechanisms of the systems under consideration. We assume that the state of a system and its environment at a single moment in time can, at least in theory, be described by a SOTA *state space*, and we call all possible trajectories the system can take through the state space over time its *trajectory space* or *phase portrait*. A more in-depth discussion of the GEM model underlying these notions can be found in [HW11], but the details are not important for understanding this pattern.

We classify awareness mechanisms along three different axes: *expressivity*, *quality* and *interface* with the rest of the system.

Expressivity. There are various types of models used in software engineering and artificial intelligence, and many systems use several types of models in different parts of their awareness models. However, most models can be classified along the following three dimensions:

Scope: To remain manageable, an awareness model will only include some dimensions of the state space, and it will only contain a limited amount of historical information. The *scope* σ of the model describes which subspace of the trajectory space is represented in the awareness model and the granularity with which this subspace is represented.

Depth: Following [KF87], we call a measure for the amount of information explicitly contained in a model that is related to the model's scope its *depth*.

Note that scope and depth are defined with relation to the state space; both "M_1 has larger scope than M_2" and "M_1 is deeper than M_2" mean that M_1 contains more information than M_2, the difference is whether this information is part of the system's state or whether it is meta-information about the model's content. Intuitively, the scope of an awareness model M describes how big the slice of the world represented by M is, and the depth of M describes how rich the ontology of the model is.

As an example, we may look at a video camera that records a room in which persons A, B and C are moving around. We assume that we are interested in the locations of the three persons, hence our state space contains (x, y) coordinates for A, B and C. If the video camera stores an hour of video, what is the scope of its awareness model? Since the video feed contains no data about the position of either person, its intersection with the state space, and hence its scope, is empty.

A person watching the video might be able to extract information about the locations, but that is a result of the awareness mechanism of the human, not the camera. This example demonstrates that the *expressivity of a model* is not a measure for the information that can be extracted from the model by a sufficiently intelligent observer, but only for the data that is explicitly stored in the model. To put it more succinctly, expressivity of models is not equal to amount of data. If we assume that we have a smart camera that can recognize people (but not individuals) and store the information about their locations in addition to the video feed, the scope of the awareness model is no longer empty. In this case the granularity of the model is relatively low, since it cannot distinguish permutations of the locations of A, B and C. If we additionally equip the camera with a facial recognition module, the model becomes more fine-grained, since observations that were previously equivalent can now be distinguished.

As this example shows, the expressivity of the model on its own is not sufficient to describe the expressivity of a system's awareness mechanism; we must also take into account the capabilities of the reasoning engines. For example, if the smart camera in the previous example stores only the image data, the model together with the recognition module can still provide information about the location of A, B and C, even though this information is not explicitly stored in the model.

The two dimensions discussed above, scope, and depth, can also be used to characterize the entire awareness mechanism if we generalize them from the data stored in the awareness model to the questions that can be asked of the awareness mechanism and the answers it provides. There are some technical challenges in providing precise definitions, but their intuitive meaning remains unchanged: the *scope* of an awareness mechanism describes the slice of the world (i.e., trajectory space) about which the awareness mechanism can provide answers and the amount of detail provided by the answers, and its *depth* the ontological and structural complexity of questions and answers.

Quality. An awareness model that has great scope, breadth and depth, yet bears only a remote relationship to the actual environment in which a system operates is not particularly useful. Similarly, a reasoner that can answer a wide range of questions may not be useful for a system if it takes too long to derive answers. Therefore we are not only interested in the expressivity of awareness mechanisms but also in their *quality* which we subdivide into accuracy and performance:

Accuracy: The *accuracy* of an awareness mechanism is a measure for the distance between answers provided by the awareness mechanism and the corresponding "real" values. This measure also takes into account the different states of the system, e.g., stigmergy-based awareness might only be able to access information stored in currently visible parts of the environment and therefore the accuracy of awareness might strongly depend on the physical location of the system or its nodes. In dynamic environments, accuracy depends on how frequently the awareness model is updated.

Performance: We define the performance of an awareness mechanism as a measure of the probability that a set of queries having a particular maximum level of complexity can be answered with a certain minimal level of accuracy in a given time.

The accuracy of different awareness mechanisms can be estimated by comparing the data in the awareness model with reality, e.g., by measuring the total difference between the location of a robot R obtained from its awareness model and its actual location (as observed by a high-precision tracking system) over the duration of a simulated rescue mission. In dynamic environments, accuracy obviously depends on the time needed to update R's awareness model after changes in the environment. For example, assume that R has a map of the environment that it uses for path planning, but that R only updates its models using data from its own sensors. If an avalanche blocks a part of the road that R

intends to take, this will not be reflected in R's model until R reaches the blocked part of the road, so this aspect of the model is inaccurate over long periods of time. If, however, R also updates its model based on information received from other robots it may increase the accuracy of its awareness since data about remote obstacles can be integrated into the model in a timely manner.

For some reasoning engines it is possible to increase the performance of inferences by reducing the accuracy of the answers. For example, reasoners that rely on local search or Monte-Carlo simulations can often control the number of iterations they perform and thereby trade accuracy for performance. Some awareness-mechanisms need to be provided with a time limit before the query is processed, others can provide a result whenever it is requested, with longer waiting times leading to improved accuracy. In analogy to the terminology for algorithms we call the latter *anytime* awareness mechanisms.

Since many reasoning mechanisms are not completely deterministic, the quality of an awareness mechanism is best described as the probability that a query with a certain level of expressivity can be answered with accuracy a in time t, i.e., quality is not a function, but rather a probability distribution over accuracy and performance conditioned on the expressivity of the queries (and other factors, such as the allocated time and resources for reasoning).

Interface. A third aspect that distinguishes different awareness mechanisms is how much access they permit for the rest of the system or to external observers, and how their connection with the rest of the system is achieved. We usually use the term *interface* to describe the connection that an awareness mechanism has to the rest of the system. In software-intensive system the awareness mechanism may provide and require precisely specified interfaces to interact with other components. In biological systems the connection between the awareness mechanism and the rest of the system is often much less clearly delineated and corresponds better to the notion of *combination operator* from [HW11]. The interface may not provide the full expressivity and quality of the awareness mechanism to the rest of the system. For example, the interface of an intelligent camera might only expose aggregate data and not the locations of individual persons at particular times. We call these aspects of the interface of an awareness mechanism its *accessibility*.

Degree of Awareness. With the previously discussed dimensions of awareness mechanisms, a *non-operational* (or *structural*, *white-box*) classification of degrees of awareness is relatively straightforward: The *(internal) degree of awareness* of a system is determined by the expressivity (scope, granularity and depth), and quality (performance, accessibility) of its awareness mechanism. We call the integration of the awareness mechanism the *(structurally) exposed degree of awareness* of a system. Various (non-operational) notions of self-awareness found in the literature can be expressed in our classification by placing constraints on the expressivity of the awareness mechanism.

Implementation

In the ASCENS project, the jRESP [BHK+12] implementation of SCEL [DLPT13] is typically used to define the overall behavior of the ensemble, the communication and coordination between components and the overall behavior of individual components. The awareness mechanism of each component performs reasoning tasks for the SCEL controller of this component; it can be implemented, e.g., in *KnowLang* [Vas12a] using the *KnowLang* reasoner or in POEM [Höl13] using the Iliad/jRESP integration. The sensor system of the awareness mechanism can either be part of the SCEL program, or it can be performed transparently by the *KnowLang*/POEM runtime. One possible way to structure the subsystems of an awareness mechanism is as *pyramid of awareness* [Vas12b], see Fig. 3.

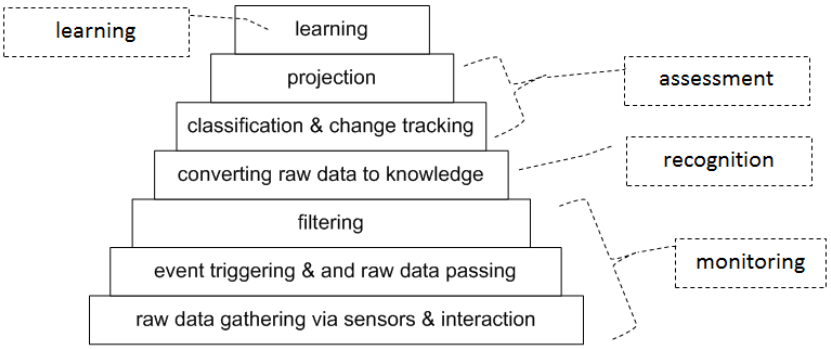

Figure 3: Pyramid of awareness

To demonstrate the interaction between SCEL and an awareness mechanism we show a simple example in which a jRESP controller uses a knowledge repository that is backed by an ILIAD reasoner. In this example a single robot has to navigate from its current position in a grid world to a given target position. The SCEL program controls the movement of the robot; to determine the next action it puts information about the current state to the POEM knowledge repository and then gets the next action from this repository. By being responsible for the update of the knowledge repository, part of the SCEL program is part of the awareness mechanism.

The class `ScelPoemNav` implements this behavior in jRESP. The attribute `poem` represents the POEM knowledge repository; tuples and templates for this repository are created by the utility functions `makePoemTuple` and `makePoemTemplate`; as usual the methods `put` and `get` are used to store and retrieve tuples. The method `getPoemValue` is a simple wrapper around `get` that casts the result from type `Object` to the more specific type `PoemValue`. The method `navigateToTarget` initially informs the reasoner about the current position and the target position by placing two tuples $\langle \textit{set-current-pos}, \textit{cur}_x, \textit{cur}_y \rangle$ and $\langle \textit{set-target-pos}, \textit{target}_x, \textit{target}_y \rangle$ in the `poem` knowledge repository. It then repeatedly queries the repository for the next move by calling `get` with the template $\langle \textit{get-next-move} \rangle$ on the repository. This operation returns the compass direction of the next move, or the value `NIL` if the robot has reached the target. When a compass direction is returned, the controller executes the requested move by driving to the new location and then informs the reasoner about the new position and the cost for performing the action.

```
public class ScelPoemNav {
  public ScelPoemNav(int curX, int curY, int maxMoves) {
    this.maxMoves = maxMoves;
    this.curX = curX;
    this.curY = curY;
  }

  // ...
  private PoemAdaptor poem;
  private int maxMoves;
  private int curX;
  private int curY;

  public void navigateToTarget(int targetX, int targetY)
        throws InterruptedException {
    poem.put(makePoemTuple("set-current-pos", curX, curY));
    poem.put(makePoemTuple("set-target-pos", targetX, targetY));
    PoemValue nextMove;
    for (int i = 0; i < maxMoves; i++) {
```

```
      nextMove = poem.getPoemValue(makePoemTemplate("(get-next-move)"));
      if (nextMove.equals(PoemSymbol.NIL)) {
        break;
      }
      executeMove(nextMove);
    }
  }

  private void executeMove(PoemValue nextMove)
        throws InterruptedException {
    int movementCost = driveToNewPosition(nextMove);
    poem.put(makePoemTuple("set-current-pos", curX, curY));
    poem.put(makePoemTuple("set-action-cost", movementCost);
  }

  private void driveToNewPosition(PoemValue nextMove) {
    // Drive to new position and update curX and curY.
  }
}
```

The definitions of `makePoemTuple` and `makePoemTemplate` determine the behavior of the reasoner when a `put` or `get` request is sent to the reasoner. For this example, each request simply calls a corresponding function in the reasoner, so that a basic implementation of the awareness mechanism could be achieved as follows:

```
(defstruct pos
  (x 0 :type integer)
  (y 0 :type integer))

(defvar *current-pos* (make-pos))
(defvar *target-pos* (make-pos))
(defvar *action-cost* 0)

(defun set-current-pos (x y)
  (setf *current-pos* (make-pos :x x :y y)))

(defun set-target-pos (x y)
  (setf *target-pos* (make-pos :x x :y y)))

(defun set-action-cost (cost)
  (setf *action-cost* cost))

(defun get-next-move ()
  ;; Compute and return the next move, e.g., by learning a map of the environment.
  )
```

Depending on the details of the scenario the function `get-next-move` could, e.g., use the reinforcement learning mechanism of ILIAD to learn how to navigate in an unknown environment.

5 Conclusions and Work Planned for Reporting Period 4

Work package 8 has made significant progress in the third reporting period: the Ensemble Development Life Cycle serves to structure the numerous activities of the ASCENS project in a way that clearly illustrates the dependencies between different research areas in the project and possible workflows for development processes using ASCENS results. The catalogue of patterns reinforces this structure by

providing links between different phases and activities, showing ways for reducing larger tasks to simpler activities and detailing a number of best practices. An important function of the pattern catalogue is also to serve as a tool for learning about different aspects of the development process.

In the fourth reporting period we will continue to work on the EDLC, the pattern catalogue and the service-component repository with particular focus on the following topics

- We will refine and further detail the EDLC using the experience gained from applying it to the case studies.
- We will complete the unfinished patterns in the pattern catalogue and, continuing the close collaboration with work package 7, identify new patterns that arise from applying the tools and techniques developed in the project to the case studies. In the reverse direction, we will try to apply the patterns to the case studies and evaluate their effectiveness.
- We will investigate whether the pattern catalogue and the service-component repository (SCR) can be cross-linked, e.g., by suggesting patterns that are applicable for the use of a chosen component in the SCR, or by adding links from patterns in APEX to the SCR entries for components that support them.
- Several important aspects for ensemble are currently not made explicit in the structure of the pattern catalogue. For example, many design decisions impact the security of a system or the privacy of its users. Some patterns mention these concerns in their description, but they are not explicitly addressed by the pattern language. It might be worthwhile to extend the pattern template with additional keywords for these kinds of concerns, but this carries the risk of inflating the number of optional keywords to the point where the pattern system loses its internal coherence. We will investigate possible extensions of the pattern template and experiment with the results they have on the pattern catalogue.

References

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

[Bel13] Lenz Belzner. Action Programming In Rewriting Logic (technical communication). *Theory and Practice of Logic Programming, On-line Supplement*, 2013.

[BHK+12] Tomas Bures, Vojtech Horky, Jaroslav Keznikl, Michele Loreti Jan Kofron, and Frantisek Plasil. Language Extensions for Implementation- Level Conformance Checking. ASCENS Deliverable D1.5, October 2012.

[DLPT13] Rocco De Nicola, Michele Loreti, Rosario Pugliese, and Francesco Tiezzi. SCEL: a Language for Autonomic Computing. Technical report, Univ. Firenze, 2013. `http://rap.dsi.unifi.it/scel/pdf/SCEL-TR.pdf`.

[GHJV95] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Longman, Amsterdam, 1995.

[HBGK13] Matthias Hölzl, Lenz Belzner, Thomas Gabor, and Annabelle Klarl. D8.2: Second Report on WP8: The ASCENS Service Component Repository (first version), 2013.

[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.

[KBC+13] N. Koch, T. Bures, J. Combaz, A. Lluch Lafuente, R. De Nicola, S. Sebastio, F.Tiezzi, A. Vandin, F. Gaducci, U. Montanari, M. Hölzl, A. Klarl, P. Mayer, M. Loreti, C.Pinciroli, M. Puvani, F. Zambonelli, and N. Šerbedžija. Software Engineering for Self-Aware SCEs: Ensemble Development Life Cycle, 2013.

[KF87] David Klein and Timothy W. Finin. What's in a Deep Model? A Characterization of Knowledge Depth in Intelligent Safety Systems. In John P. McDermott, editor, *IJCAI*, pages 559–562. Morgan Kaufmann, 1987.

[LDB10] Axel Legay, Benoît Delahaye, and Saddek Bensalem. Statistical Model Checking: An Overview. In Howard Barringer, Yliès Falcone, Bernd Finkbeiner, Klaus Havelund, Insup Lee, Gordon J. Pace, Grigore Rosu, Oleg Sokolsky, and Nikolai Tillmann, editors, *RV*, volume 6418 of *Lecture Notes in Computer Science*, pages 122–135. Springer, 2010.

[Puv12] M. Puviani. TR 4.2: Catalogue of Architectural Adaptation Patterns. Technical report, ASCENS Project, 2012.

[Vas12a] E. Vassev. Operational Semantics for KnowLang ASK and TELL Operators. Technical Report Lero-TR-2012-05, Lero, University of Limerick, Ireland, 2012.

[Vas12b] Emil Vassev. Building the Pyramid of Awareness. *Awareness Magazine - Self-awareness in Autonomic Systems*, 07/2012 2012.

[WHTZ11] Martin Wirsing, Matthias M. Hölzl, Mirco Tribastone, and Franco Zambonelli. ASCENS: Engineering Autonomic Service-Component Ensembles. In Bernhard Beckert, Ferruccio Damiani, Frank S. de Boer, and Marcello M. Bonsangue, editors, *FMCO*, volume 7542 of *Lecture Notes in Computer Science*, pages 1–24. Springer, 2011.

[ZAC+12] Franco Zambonelli, Dhaminda B. Abeywickrama, Giacomo Cabri, Mariachiara Puviani, Matthias Hölzl, Andrea Corradini, Alberto Lluch Lafuente, and Rocco De Nicola. D4.2: Second Report on WP4. Component- and Ensemble-level Self-Expression Patterns: Report on Experimental and Simulation Activities, and requirements for Tools Implementation, October 2012.