ascens
www.ascens-ist.eu

ASCENS

Autonomic Service-Component Ensembles

JD1.1: Engineering Ensembles
A White Paper of the ASCENS Project

Grant agreement number: **257414**
Funding Scheme: **FET Proactive**
Project Type: **Integrated Project**
Latest version of Annex I: **7.6.2010**

Lead contractor for deliverable: **LMU**
Author(s): **Matthias Hölzl, Martin Wirsing, Annabelle Klarl, Nora Koch, Stephan Reiter, Mirco Tribastone (LMU), Rocco De Nicola (IMT), Diego Latella, Mieke Massink (ISTI), Ugo Montanari, Roberto Bruni (UNIPI), Lubomír Bulej, Jan Kofroň (CUNI), Joseph Sifakis, Saddek Bensalem, Jacques Combaz (UJF-Verimag), Emil Vassev (UL), Franco Zambonelli (UNIMORE)**

Due date of deliverable: **September 30, 2011**
Actual submission date: **October 19, 2011**
Revision: **V1 Final**
Classification: **PU**

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

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

Executive Summary

Today's developers often face the demanding task of developing software for ensembles: systems with massive numbers of nodes, operating in open and non-deterministic environments with complex interactions, and the need to dynamically adapt to new requirements, technologies or environmental conditions without redeployment and without interruption of the system's functionality. Conventional development approaches and languages do not provide adequate support for the problems posed by this challenge.

The goal of the ASCENS project is to develop a coherent, integrated set of methods and tools to build software for ensembles. To this end we research foundational issues that arise during the development of these kinds of systems, and we build mathematical models that address them. Based on these theories we design a family of languages for engineering ensembles, formal methods that can handle the size, complexity and adaptivity required by ensembles, and software-development methods that provide guidance for developers.

Contents

1 Introduction **5**
1.1 Ensembles 5
1.2 The ASCENS Approach 5
1.3 ASCENS Languages for Engineering Ensembles 7
1.4 The ASCENS Case Studies 7
1.4.1 Ensembles of Self-Aware Robots. 8
1.4.2 Resource Ensembles as Science Clouds. 8
1.4.3 Ensembles of Cooperative E-Vehicles. 8

2 Foundations: Adaptation, Awareness, Knowledge, Emergence **9**
2.1 GEM: The General Ensemble Model 9
2.2 Adaptation 9
2.3 Awareness 10
2.4 Knowledge 10
2.5 Emergence 11

3 Structure: Service-Component Ensembles **12**
3.1 Service Components 12
3.2 Service-Component Ensembles 14

4 Design: Languages and Models **14**
4.1 Declarative Modeling Languages: KnowLang, SOTA, POEM 14
4.2 SCEL: A Service Component Ensemble Language 15
4.3 Dynamic BIP – Behavior, Interaction, Priority 16
4.4 Foundational Models 17
4.5 Performance Models 18

5 Validation and Verification: Formal Methods **19**
5.1 From A Posteriori Verification to Constructivity 19
5.2 Properties of Implementations 21

6 Engineering Ensembles **21**
6.1 Best Practices and Patterns 21
6.2 Tool Support 22

7 Conclusions **22**

1 Introduction

In the beginning, things were not going well at all. The swarm of miniature robots had only been deployed for an hour and already more than one third of the robots had become incapacitated by getting stuck in locations from which they could not extricate themselves. The robots were of course equipped with terrain sensors and control logic designed to prevent this, but a combination of unexpected terrain and lighting conditions caused the algorithm that had worked reliably during testing to produce disappointing results. Realizing that they were in danger of failing in their task to locate rare-earth metals in this remote area, the robots changed their behavior: Instead of foraging individually the robots used their grippers to connect themselves into larger structures that were better suited to explore the assigned region without becoming trapped, thereby trading off speed of exploration for increased reliability. Using this strategy they even managed to free half of the immobilized robots while those robots that could not be extracted reconfigured themselves as communication relays, allowing robot groups in their vicinity to disperse further without being in danger of losing contact with each other. After several hours of foraging the swarm discovered a large deposit of monazite sand and mapped its location, making the mission a success.

Unfortunately, current technology is not sufficiently advanced to build a robot swarm that actually exhibits the behavior described in this scenario. While it is already possible to design hardware that has the required features, our current approaches to software engineering are not sufficient for building systems that can autonomously adapt to a wide range of different, unexpected situations based on awareness of the environment, the system and its goals, as the robot swarm in the scenario does. The aim of the ASCENS project is to develop foundations, techniques and tools to engineer software for this kind of system.

1.1 Ensembles

Numerous reasons why it is necessary to develop large software-intensive systems with the capabilities to operate in unknown environments have been documented [HRW08]. The ICT-FET project InterLink [Int] has coined the term *ensemble* for a particular kind of system: Ensembles are software-intensive systems with massive numbers of nodes or complex interactions between nodes, operating in open and non-deterministic environments in which they have to interact with humans or other software-intensive systems in elaborate ways. Ensembles have to dynamically adapt to new requirements, technologies or environmental conditions without redeployment and without interruption of the system's functionality, Thereby blurring the distinction between design-time and run-time.

Of course, systems that satisfy the definition of ensembles have already been built: National infrastructures such as the power grid, large online businesses such as Amazon or Google, or the systems used by modern armies, all satisfy the definition of an ensemble. However, these systems solve relatively well-understood problems and their size is mostly a function of the amount of data and the number of transactions they have to process. These are interesting problems in themselves, but not the main complication for building the kind of ensemble described in the initial scenario: None of these systems can actually adapt to unforeseen environmental conditions in the same way as our hypothetical robot system.

1.2 The ASCENS Approach

Instead of static software that operates without knowledge about its environment and hence relies on manual configuration and optimization we have to build systems with self-aware, intelligent components that mimic natural features like adaptation, self-organization, and autonomous as well as collective behavior. However, traditional software engineering, both agile and heavyweight, relies to a large

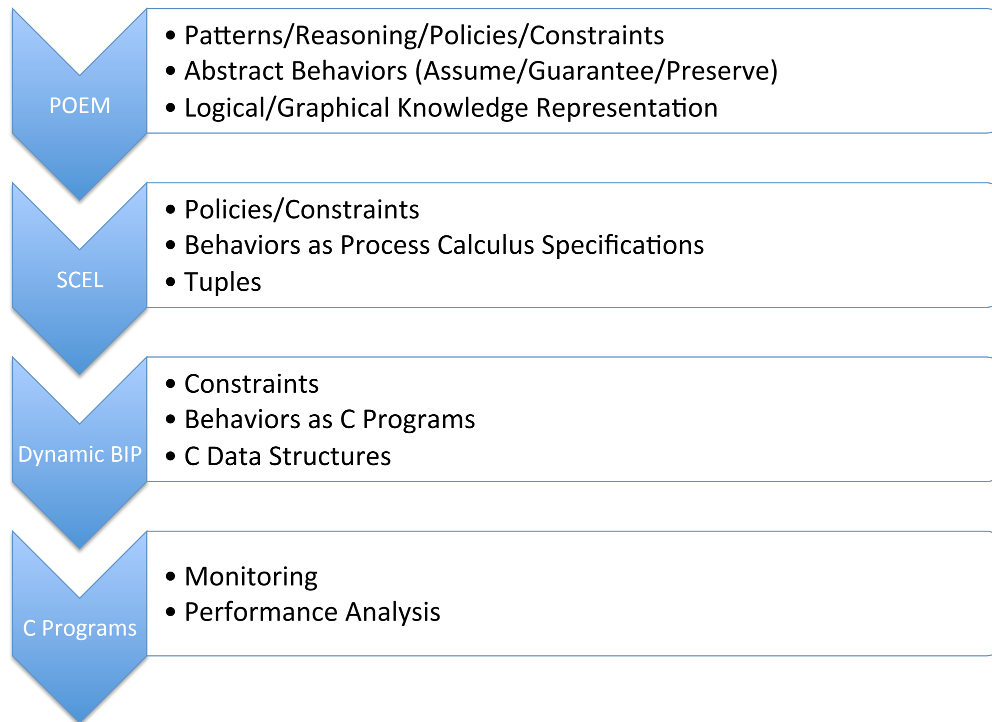

Figure 1: Relationship between the behavorial ASCENS languages

degree on code inspection and testing, approaches which are not adequate for reliably developing large concurrent systems, let alone self-aware, adaptive systems. Formal methods have successfully been employed in an ever increasing number of projects; however, they generally cannot deal with the dynamic and open-ended nature of the systems we are interested in, and they are difficult to scale to the size of industrial-scale projects. Approaches from autonomic and multi-agent systems address aspects such as self-configuration and self-optimization, but they lack necessary guarantees for reliability, dependability and security and are therefore not appropriate for critical systems.

The goal of the ASCENS project is to build ensembles in a way that combines the maturity and wide applicability of traditional software-engineering approaches with the assurance about functional and non-functional properties provided by formal methods and the flexibility, low management overhead, and optimal utilization of resources promised by autonomic, self-aware systems. To this end we are researching and inventing new concepts for the design and development of autonomous, self-aware systems with parallel and distributed components. We are developing sound, formal reasoning and verification techniques to support the specification and development of these systems as well as their analysis at run-time. The project goes beyond the current state of the art in solving difficult problems of self-organization, self-awareness, autonomous and collective behavior, and resource optimization in a complex system setting.

To enable the specification and analysis of ensembles at various levels of abstraction the ASCENS project has defined a denotational system model for ensembles called *General Ensemble Model (GEM)*. GEM is a mathematically precise model for ensembles that is useful for defining properties such as adaptation, awareness and emergence. Other foundational models developed as part of ASCENS provide precise mathematical semantics for aspects such as the network architecture of an ensemble.

1.3 ASCENS Languages for Engineering Ensembles

GEM itself is not directly usable as an engineering language, but it serves as a semantic foundation for the various logics, models and languages defined in ASCENS to support the development of ensembles. For the mostly declarative description of ensembles three closely related languages and models are defined: *KnowLang* for modeling knowledge and inference in the ensemble and its individual components, *State Of The Affairs (SOTA)* for modeling adaptation strategies and patterns, and the *Pseudo-Operational Ensemble Model* language *(POEM)*, a declarative language for expressing (not necessarily executable) behavioral models and goals. Operational models of ensembles are expressed in the *Service-Component Ensemble Language (SCEL)*, a flexible process-calculus inspired language. At a level even closer to the actual machine code, *Dynamic BIP (Dy-BIP)* can be used to express the architecture of ensembles in an expressive graphical notation while the code and data structures are specified in plain C code. The development process can progress from POEM models to SCEL specifications which can either be automatically or manually translated into Dy-BIP programs. The BIP code generator can produce a C program from the Dy-BIP specification. Powerful tools for formal analysis will be available for all languages and integrated into a development environment; the run-time behavior of programs can be monitored and analyzed using the ASCENS monitoring and performance analysis tools. Fig. 1 presents this process graphically; more detailed descriptions of the languages can be found in Sect. 4.

1.4 The ASCENS Case Studies

While one of the main goals of ASCENS is the development of foundational theories of ensemble engineering, we are also deeply concerned with the applicability of our results. Therefore, the fundamental research is closely integrated with application-oriented case studies that provide continuous feedback. To ensure wide applicability of our results we have chosen three case studies from different areas: Swarm robotics, cloud computing and e-mobility.

Figure 2: Robot swarm on a simulated rescue mission

1.4.1 Ensembles of Self-Aware Robots.

The swarm-robotics case study considers ensembles of cooperating, self-aware robots. Robot swarms generally consist of relatively cheap robots that lack many of the capabilities of larger, more sophisticated models, but that can collaborate to collectively achieve tasks that no individual robot in the swarm could accomplish. Robot swarms are particularly well suited for operations in difficult environments where the risk of failure for individual robots is high. For example, robot swarms might be used to rescue victims of natural disasters or industrial accidents. In these situations, the attrition rate of individual robots may be high, but a swarm of robot can continue to function even when individual robots have failed.

In ASCENS we will mainly investigate scenarios where the goal of the robot swarm is to localize and transport objects; this task is similar to the one presented in the initial scenario and also to the rescue task. The objects will be defined in such a way that their transport requires physical cooperation of several robots. This task will have to be performed in a number of increasingly challenging environments containing obstacles such as holes, hills, barriers, bridges, slopes, etc.

1.4.2 Resource Ensembles as Science Clouds.

The Science Cloud case study is about making cloud computing more dynamic and open while attempting to maintain its properties of being a reliable and flexible approach for using third-party resources and services, something that is done by both companies and private users using commercial and in-house clouds.

Our goal is to create a platform-as-a-service (PaaS) solution for data sharing and execution of distributed applications, which is based on autonomous, cooperating computers that provide their storage and computational resources on a best-effort and at-will basis. In particular, computers contributing to this platform are not required to stay available for any pre-determined amount of time and may join or leave the system at any given moment. We will explore this property in the form of ad-hoc cooperations between researchers from different organizations. Mechanisms to deal with transiently available computers are a requirement for a usable platform and will therefore be in the focus of the case study, resulting in a highly fault-tolerant solution that will also yield better services compared to those delivered from traditional environments, such as data centers which typically have to deal with failing servers.

1.4.3 Ensembles of Cooperative E-Vehicles.

The e-mobility case study aims at illustrating the theories and methodologies developed in ASCENS in the domain of e-mobility planning. Driver, vehicle and infrastructure are considered as interacting autonomous Service Components, which are temporally re/organized in Service Component Ensembles to reach a goal. In order to provide a user with a seamless daily travel plan, a sequence of destinations with possibly different travel modes and resource requirements have to be scheduled. The main intention is to provide benefits for the individual vehicle and its driver as well as for the whole ensemble of vehicles.

Particular attention is paid to the performance criterion of providing a high-level Quality of Service (QoS) that incorporates the following features: Reliability (e.g. transport/delivery reliability, adherence to schedules, guarantee to reach the goal, recharging-in-time assurance), adaptation to changes (e.g. traffic flow, daily personal schedule of the driver) and predictability (confidence in reaching a desired location at a requested time).

2 Foundations: Adaptation, Awareness, Knowledge, Emergence

While the meaning of concepts such as adaptation, awareness, knowledge and emergence might seem to be intuitively clear, a large variety of incompatible and contradictory definitions of these terms exist in the literature. However, none of the existing definitions of these terms is simultaneously comprehensive, abstract and precise enough to serve as a formal basis for the calculi and validation techniques of ASCENS.

2.1 GEM: The General Ensemble Model

In order to provide a shared understanding and a precise mathematical semantics of the fundamental notions in the project, we are defining the General Ensemble Model (GEM), a formal system model for ensembles [HW11]. GEM is denotational in the sense that it represents ensembles as relations that describe the complete behavior of the ensemble over its lifetime. To model the hierarchical nature of ensembles, relations in GEM can be composed from simpler relations by means of combination operators. Furthermore, GEM allows the seamless integration of knowledge as well as the internal states of components to provide a semantic foundation for KnowLang, SOTA and POEM.

In order to define requirements and goals for ensembles, GEM provides a general mechanism for connecting different logics to the relational description of an ensemble, so that different formal methods can be applied to GEM models.

In addition to serving as a denotational semantics for the ASCENS languages, GEM allows us to precisely define adaptation, awareness, knowledge and emergence, and to investigate their fundamental properties independent of any operational mechanism. The next sections will provide a short introduction to these notions.

2.2 Adaptation

The behavior of the robot swarm in the initial example clearly exhibits adaptation: in spite of being placed in an unknown environment that exhibited features not anticipated during the development of the ensemble, the swarm was capable of satisfying its goal of exploring the area and finding deposits of rare-earth metals. The swarm's success was achieved by changing the structure of the swarm (e.g., instead of individually foraging, the robots decided to form larger clusters acting in a coordinated way). To achieve this the individual robots had to change their individual behaviors and goals in a coordinated manner: groups of several robots have to agree to build a cluster. Robots that have become stuck have to recognize that they can no longer achieve their original goal and find new goals that serve the overall purpose of the swarm.

These different views are a typical endeavor in software engineering, i.e., that of distinguishing between the "what" a system should do and the "how" a system should be architected. By adopting the same point of view in the analysis of adaptivity for ensembles, we can define two notions of adaptation:

- *Black-box adaptation:* The focus is on the observable adaptivity of a system, which shows itself capable of achieving application goals in a flexible yet robust way.
- *White-box adaptation:* The focus is on behavioral and structural changes, at the level of individual components, that enable the system to exhibit adaptability.

These two perspectives can be very useful to firstly assess the degree of adaptivity to situations that a system should exhibit, and to consequently understand which specific mechanisms (e.g., as chosen among a catalog of adaptation patterns [CPZ11]) one should adopt to achieve such adaptivity.

In particular, we emphasize that black-box adaptation introduces the notion of an adaptation space—essentially a set of pairs of environments and goals—and introduces the concept "ensemble E can adapt to all elements of adaptation space $\mathcal{A}$", i.e., for each pair consisting of an environment η and a goal γ in $\mathcal{A}$, the ensemble E can achieve goal γ when operating in environment η. This allows us to compare the range of situations to which various ensembles can (or has to) adapt, and can help architecting ensembles accordingly.

An important generalization is achieved by extending the binary predicate "goal satisfaction" to the more relaxed notion of "utility" or "fitness for a purpose": While goal satisfaction allows us to classify ensembles only into those satisfying a goal and those not satisfying this goal (in a given situation), it is also important to express "how well" an ensemble can satisfy certain goals. We call an ensemble together with a function that computes its utility or fitness value a *heterostatic ensemble*. Another important generalization is from the purely relational view of ensembles to a probabilistic view that describes how likely the different behaviors of the ensemble are.

Clearly, it is important to extend the above definitions of black-box and white-box adaptation so as to account for heterostatic and probabilistic ensembles, i.e., coupling the concept of goals with a concept expressing quality requirements for reaching such goals in a probabilistic setting. Another important aspect is the definition of mechanisms to dynamically "adapt adaptation quality".

2.3 Awareness

To recognize that they are in danger of failing their mission, the robots have to possess awareness of their environment: To realize that they have become stuck, they must determine that there is a large difference between the expected result of their action and the actual result; e.g., spinning the wheels should lead to movement in a certain direction, but did not influence their position at all. To evaluate whether they are acting in a manner that is helpful for reaching the ensemble's requirements, the individual robots need to be aware of the goal of the ensemble and whether their actions contribute toward reaching that goal.

To introduce a more precise definition of awareness we have to extend the the basic ensemble model beyond the purely behavioral view of the ensemble and add an internal state to the relation that describes the ensemble. Note that this does not require any changes to GEM, it suffices to introduce the internal state into the relation describing the ensemble (or one of its components). We can then say that the ensemble (or component) E is aware of another part of the system or the environment R, if the distance between the internal state of E and the representation of R in the relation is sufficiently small.

Therefore, in order to quantify awareness, it is necessary to introduce a distance measure between the values of the internal state E and the values of R.

Given this notion of awareness, self-awareness of a component can easily be described as well: it is simply the distance of the components's internal state to the whole relation describing the component. In this way we obtain a uniform model for environmental awareness, network awareness, self-awareness and ensemble awareness.

The analysis of adaptation from the black-box viewpoint can help designers identify those specific models of the environment η and the ensemble's internal behavior I that are useful for the ensemble (or an individual component) E to better achieve its goals. In other words, this analysis can identify the kind of awareness of η and I that E needs in order to adaptively achieve its goals [ZBC$^+$11].

2.4 Knowledge

In the example, the robots received signals from their sensors from which they eventually concluded that their actions would not achieve the desired goal. They could then diagnose the problems and

eventually find a different behavior that may lead to a successful outcome. In order to achieve these results, the robots have to transform the raw data into knowledge: Structured information that provides meaning, abstractions, and ways to reason about and act upon the provided data [ITE09].

A single component gains raw data about the external world by taking measurements with its sensors. However, this data is not sufficient to enable control mechanisms other than simple condition-action-rules. In order to create a solid basis for decision making, the raw data must be transformed into knowledge, i.e. it must be structured into meaningful abstractions which interpret the current measurements within the component's full context. The relevant context typically does not only consist of information about the current state of the component itself, but also includes the whole history of observations taken so far, information about the other components in the ensemble and environmental circumstances. The integration of knowledge from other components of the ensemble may also compensate uncertainty of observations or allow coordination within the ensemble. Altogether, these requirements call for appropriate mechanisms and formal notations to provide the component with means to organize its observations into knowledge elements, integrate them into a complex knowledge base and finally perform efficient reasoning to derive suitable decisions.

Many techniques for knowledge representation have been proposed, such as rules, frames, semantic networks and concept maps, ontologies, and logic expressions. In the ASCENS project, we apply the approach of ontologies to the domain of ensembles. An ontology models the world in terms of *concepts* representing meaningful units of information within a domain. Relations between these concepts capture properties and relationships between concepts. By mapping raw data to these concepts, a component can organize its knowledge. Using ontologies, a component is able to reason about its knowledge and to infer conclusions from it. If the component is missing some important data, it might be able to deduce certain facts from its knowledge by using inference mechanisms.

We model four distinct types of knowledge: *SC knowledge* considers all information about the component itself while *SCE knowledge* takes the whole ensemble into account. Thereby, a component can compare its own SC knowledge base with other components to verify or update its observations and beliefs. With the aid of the SCE knowledge, the component can understand even more about the progress and steps of the whole ensemble. Information about the operational environment is captured in the *context knowledge*. This enables the component to relate its knowledge to the environmental circumstances. At last, the component is able to distinguish between known and unknown situations by evaluating *situational knowledge patterns*. The component builds up a history of situations, actions and effects. On the basis of these "lessons learned", it characterizes different situations and the required actions as patterns and uses them to perform better in known situations.

Regardless of which kind of knowledge we consider, the knowledge must be structured so that it can be effectively and efficiently processed by an intelligent system and perceived or updated by humans. So in the ASCENS project, we emphasize the use of ontologies to do the splits between automatic processing and human readability.

2.5 Emergence

The term *emergence* has been used to describe various phenomena: in the software engineering literature it is often used to describe global phenomena, not arising from any single component [Som07]; in the literature about complex system it is often used with more specific denotations, for example Mark A. Bedau defines *weak emergence* as [Bed97]: "Macrostate P of [a system] S with microdynamic D is weakly emergent iff P can be derived from D and S's external conditions but only by simulation." In this section it is mostly this latter denotation of emergence that we are concerned with.

Emergent phenomena often occur due to the pattern of non-linear and distributed local interactions between the elements of a system over time. Surprisingly, agent-based crowd models, in which the

movement of each individual follows a limited set of simple rules, often reproduce quite closely the emergent behavior of crowds that can be observed in reality.

Consider for instance the scenario of self-aware robots and suppose their working environment be composed of several, interconnected sub-areas. One would like to see the robots evenly distributed among the sub-areas, but it may happen that due to some mutual fit parameter each robot may probabilistically decide to remain in a certain sub-area or to move away from it to another sub-area. Such a simple rule could result, over time, in widely varying distribution dynamics of the robots in the environment, ranging from even distribution, with robots smoothly moving from one sub-area to another, carrying material around, to situations in which all robots gather in a single sub-area, getting stuck there. The emergent behavior associated to such phase shifts has been studied, and formally analyzed, in the case of human agents (see, e.g. [RG03, MLBH11]) and, given the very simple local rules which govern the behavior of the agents, it is likely to be found again in robot swarms.

Another interesting scenario is that of a swarm of robots using Ant Colony Optimization Algorithms. These are optimization algorithms which are inspired by the behavior of colonies of ants hunting for food (see, e.g. [DS04]), where intelligent/optimal behavior of a large population of agents emerges from quite simple rules followed by each individual agent. The formal specification and scalable analysis of such systems, that in general consist of a large number of autonomous entities, is still a challenging problem [Tof90, SBB01], but promising results are being reached, for instance using continuous, fluid-flow, approximation techniques for stochastic process algebras [ML11] or stochastic differential equations [Mey08]. Such analysis is essential to assure functional and non-functional properties of such systems, especially when they are employed in safety critical applications.

3 Structure: Service-Component Ensembles

Compositionality is one of the most powerful tools for taming complexity: If we can structure a system into well-understood building blocks that interact in specified ways, we can reduce the complexity of innumerable interactions between low-level components to a manageable number of interactions between building blocks. The ASCENS approach therefore focuses on service-component ensembles (SCEs), hierarchical ensembles built from service components (SCs), simpler SCEs and knowledge units (K) connected via a highly dynamic infrastructure, see Fig. 3. The robot swarm demonstrates a simple hierarchical structure: each robot can be seen as an ensemble consisting of controllers, sensors actuators, etc., and the swarm of robots is an ensemble consisting of individual robots.

3.1 Service Components

Service components are nodes that can cooperate, with different roles, in possibly open and non-deterministic environments. These basic properties, already satisfied by, e.g., contemporary service-oriented architectures, will be enriched by new properties of awareness as described in section 2.3. They will allow awareness-rich behavior making SCs adaptable, connectable and composable. The self-awareness of service components in an ensemble is achieved by: (i) equipping SCs with declarative information about their own state and behavior; (ii) enabling SCs to collect and store information about their working environment, possibly gaining limited information about the whole system; and (iii) using this information for redirecting and adapting SC behaviors via the proper engineering of autonomic control loops either within or outside components.

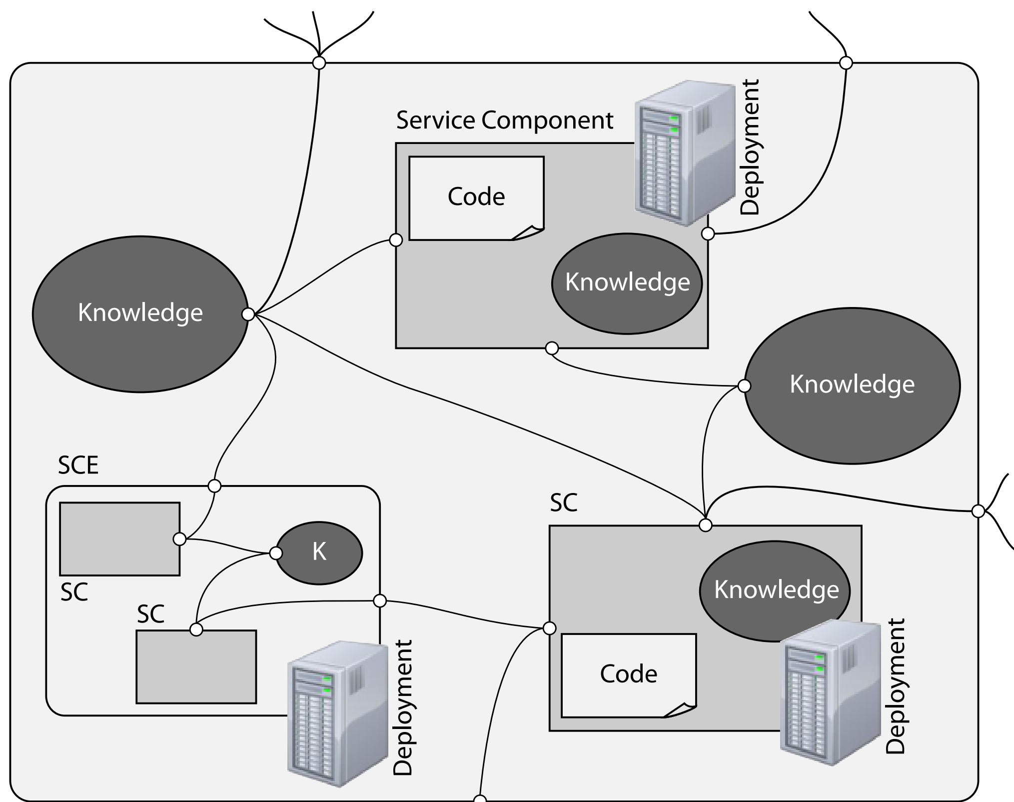

Figure 3: Service Component Ensemble

3.2 Service-Component Ensembles

A service-component ensemble is a set of service components (SCs) with dedicated knowledge units, representing the shared local and global knowledge about levels of awareness, resources, connectivity and networking, interconnected in a dynamic network, featuring goal-oriented, safe and secure execution, and efficient resource management.

To realize ensembles of service components whose properties go far beyond the state of the art in current software engineering and technology, we are thoroughly investigating the following domains:

1. Linguistic support for programming SCEs, expressing awareness and exchanging knowledge;
2. Formalization and modeling the fundamental SCE network properties like autonomous behavior and awareness-rich networking;
3. Knowledge representation and self-awareness of service components;
4. Methods and mechanisms for adaptation and dynamic self expression, possibly defined in terms of feedback loops and architectural patterns for organizing the desired adaptation;
5. Techniques and methodologies for the design and development of reliable SCs and SCEs and their verification using formal methods;
6. Software infrastructure to support programming, deployment and execution of SCE-based applications;
7. Performance prediction targeting scalable performance-analysis methods and performance monitoring at run-time;
8. Checking the compliance of SC implementation code with its formal specification.

Each of the mentioned research topics is based on the system model for ensembles presented in the previous section.

4 Design: Languages and Models

In principle the software of our hypothetical robots could be developed using traditional modeling and programming languages. However, these languages lack support for conveniently expressing the concepts presented in Sect. 2. To manage the complexity of ensemble development, developers need languages that can express these important features of the software more directly.

In traditional software engineering, modeling languages like the UML provide different sub-languages or diagram types for modeling various aspects of a program, e.g., its architecture, static structure, or the behavior of objects. Ensembles present several novel problems for the software engineer, e.g., the need to take into account more sophisticated knowledge, adaptation or the interplay between goals of individual components and the ensemble. To address these issues, ASCENS is defining a family of languages; the formal basis for these languages is provided by the foundational models described in Sect. 2.1 and 4.4. Ensembles often interact with the physical world and therefore have strict performance requirements; performance models are introduced in Sect. 4.5.

4.1 Declarative Modeling Languages: KnowLang, SOTA, POEM

Declarative models of ensembles are expressed in three closely related ASCENS languages: KnowLang for knowledge, SOTA for adaptation, and POEM for goals and behaviors.

KnowLang is a knowledge representation language that allows the definition of ontologies and knowledge bases, and it defines knowledge-base operators for maintaining and querying the knowledge base as well as inference primitives. KnowLang provides a multi-tier specification model that allows the presentation of knowledge at multiple levels of depth of meaning.

SOTA is a formalism for modeling adaptation requirements and helping the designer chose the most adequate adaptation patterns. SOTA is based on the same principal ideas as GEM, but whereas GEM is a general model of ensembles, SOTA is focused on those aspects which are relevant to adaptation, which results in a simpler and more streamlined presentation.

POEM is a specification language for behavior and goals; it is based on the same foundations as KnowLang, i.e., the event calculus and graphical probabilistic networks. POEM is closely integrated with KnowLang and SOTA; furthermore POEM models can contain SCEL programs for executable behaviors. Therefore POEM models can include both declarative and procedural behavioral models and therefore allow specifications at various levels of detail. A particular focus of POEM is the interplay between behaviors of individual components and the overall ensemble, the dynamic trade-off between reasoning and precompiled behavior, and the integration of design time and run time.

4.2 SCEL: A Service Component Ensemble Language

As mentioned above, the behavior of robots, their interactions, sensitivity to the environment, adaptivity need, of course, to be programmed. This could theoretically be done in any of the existing programming languages, even in an assembly language. However, given the intricacy of the issues under consideration and the need to foresee the emergent behavior of many interacting agents and to guarantee that specific functionalities are offered, programming the robot swarm in a traditional programming language would be so complex as to be practically infeasible. It would be better to resort to a language where notions such as component, interaction, interface, distribution, mobility, knowledge, awareness or adaptation are first class elements. This absolves the developer from the need to elaborate these constructs each time they occur in the program. To facilitate integration with the formal methods used in ASCENS, it is essential that the language is based on a solid semantic ground so that formal reasoning can be performed.

For these reasons, we are designing SCEL, a new language that brings together various programming abstractions that permit directly representing knowledge, behaviors and aggregations according to specific policies and, naturally, to program interaction, adaptation and self- and context-awareness.

The abstractions related to *knowledge* describe how knowledge is manipulated and shared. At the SCEL level, knowledge is represented through multi-sets of items containing application data and control data. The former are used for the progress of components, while the latter provide information about the environment in which components are running (e.g. monitored data from robot's sensors) or about the actual status of a single autonomic component (e.g. robot's position or remaining energy). To model distribution, each component has a private repository that may or may not (depending of the conditions) be accessed by others.

The abstractions related to *behaviors* describe how components progress and are modeled as processes in the style of process calculi. Interaction is modeled by allowing different components to access the knowledge repository of other components. Adaptation is modeled by retrieving from the knowledge repository both information about the changing context and suggestion about the code to execute.

The abstractions related to *aggregations* describe how different entities are brought together to form components, systems and, possibly, ensembles and are useful to model distribution and mobility. Compositionality and interoperability are supported by interfaces, that specify attributes and functionalities provided and/or required by components.

The abstractions related to *policies* deal with the way properties of computations are represented and enforced. Interaction and Service Level Agreement (SLA) provide two standard examples of policy abstractions. Other examples are security properties maintaining the right linkage between data values and their associated usage policies (data-leakage policies) or limiting the flow of sensitive

information from untrusted sources (access control and reputation policies).

All these abstractions are based on solid semantics grounds that lay the basis for developing logics, tools and methodologies for establishing qualitative and quantitative properties about the behavior of both the individual components and the overall ensemble.

4.3 Dynamic BIP – Behavior, Interaction, Priority

Behaviors of SCEs often arise from the individual actions of multiple SCs. This can result in a wide variety of useful properties, but also in unplanned processes which are detrimental to the assigned goals of the ensemble. In any case the possible interactions are determined by the structure of the ensemble and the possible interactions between components. Therefore architectures are essential for mastering the complexity of ensembles and to facilitate their analysis and evolution. They allow separation of detailed behavior of components and their overall coordination. Coordination is usually expressed by constraints that define possible interactions between components. For instance, robots in swarm can only interact if they are in the range permitted by their communication devices. Clearly, static architectures are inefficient for systems that exhibit adaptive behavior, e.g. the set of interactions that can eventually occur in a swarm is intractable for a large number of robots.

Dy-BIP is a component framework based on rigorous operational semantics for modeling both static and dynamic architectures at a level closer to the actual hardware than SCEL. Dy-BIP can be considered as an extension of the BIP language for the construction of composite hierarchically structured components from atomic components. These are characterized by their behavior specified as automata labeled by ports, and extended with data and functions described in C. In BIP architectures are used composition operators on components defining their interactions. An interaction is described as a set of ports from different components. It can be executed if there exists a set of enabled transitions labeled by its ports. The completion of an interaction is followed by the completion of the involved transitions: execution of the corresponding actions followed by a move to the target state. An operational semantics for BIP has been defined in [BBBS08].

In contrast to BIP, the set of interactions characterizing architectures in Dy-BIP changes dynamically with states. A port p has an associated architecture constraint C_p which describes possible sets of interactions involving p. Feasible interactions from a state are computed as solutions of the constraint obtained as the conjunction of the constraints of all the enabled transitions. We provide a formalization of the operational semantics for Dy-BIP.

A BIP model (i.e. using a static architecture) with a global architecture constraint C, can be represented as a Dy-BIP model such that the constraint C_p associated with a port p is the set of the interactions of C involving p. Dy-BIP allows modeling dynamic architectures as the composition of instances of component types. We first assumed that there is no dynamic creation/deletion of component instances.

The language for the description of architecture constraints in Dy-BIP is expressive and amenable to analysis and execution. It defines formulas of a first order logic allowing quantification over instances of component types. Formulas characterize sets of interactions. They involve port names used as logical variables. Given a formula, a feasible interaction is any set of ports assigned true by a valuation which satisfies the formula.

The semantic model and associated modeling methodology for writing architecture constraints is as follows. For a port p, the associated constraint is decomposed into three types of dependency characterizing interaction between ports [BS08]: "causal constraint", "acceptance constraint", "filter constraint". A causal constraint defines the ports required for interaction. An acceptance constraint defines optional ports for participation. Filter constraints are invariants used to exclude undesirable configurations of a component's environment.

Dy-BIP semantics is implemented in terms of an Execution Engine handling symbolic architecture constraints. As for BIP, the Engine orchestrates components by executing a three-step protocol. The protocol differs in that components send not only port names of enabled transitions but also their associated architecture constraints. The proposed implementation of the Engine is based on resolution of architecture constraints on the fly. It uses efficient constraint resolution techniques based on BDDs.

4.4 Foundational Models

In order to be useful for formal analysis and verification the languages described above have to be given a precise semantics. The foundational models developed as part of ASCENS provide the mathematical foundations for this task. GEM, the system model for ensembles, has already been described in Sect. 2.1. Other foundational models focus on more specific questions, such as the structure of the network or strategies to achieve cooperative behaviors.

As an example, we look at the communication layer between the robots in the initial scenario. Existing ad hoc and opportunistic computer networks already provide most of the features we need to model this layer at the lowest level of abstraction: taking advantage of other robots as network nodes, reconfiguring the networks according to the topology and the existing resources such as power and bandwidth, etc. At a higher level of abstraction we need more intricate behaviors, as they are, for example provided by *active networks* [DAR] which have been extensively studied in the US in the past 15 years: certain features of the communication system, e.g. the routing tables, are made accessible to, and modifiable by, the application programmer. For instance, in the above scenario, suitable communication protocols could have been uploaded to increase the bandwidth among close robots while forming the groups.

Foundational models of ASCENS will describe these abstractions at the formal level and apply the resulting models to the ASCENS languages.

To this end we are contributing an advanced network-aware model relying on a sophisticated middleware of connectors. During execution, the connectors can allocate and deallocate shared resources. As is well known in the theory of distributed systems, the key feature for consensus is the ability of agreeing on a shared value, namely to synchronize. Thus the model relies on a variety of synchronization primitives, from message reception in an asynchronous setting, to Petri net/Linda distributed decisions, to REO synchronization primitives. Synchronization means a commit on a contract signed by two or more agents, or, more generally, the solution of a local, distributed problem. After the commit, certain properties should be ensured. In the initial swarm robotics example, the commit might correspond to the formation of the groups, possibly after several trials represented as transaction backtrackings. After forming the group, the contract will guarantee that certain movements can happen without breaking the group.

Shared abstractions are the subject of a second class of foundational models. Good, well understood examples are concurrent constraints and execution-time behavioral types. In general, we typically have a trade-off between procedural and declarative knowledge: the abstractions could be values, constraints (i.e. sets of possible values) or execution rules. Several different abstractions for the same concepts may exist and the one most suitable for a purpose may be chosen dynamically at run time, for example knowledge how to achieve certain tasks may be present in declarative form but also as precompiled programs: operating on the declarative knowledge may be inefficient and consume a lot of storage and energy, but it provides a wide range of possibilities for behavioral adaptation which can be activated in situations when the precompiled programs fail.

Usually abstractions are local, namely are shared among a small number of processes. However they can be equipped with *closure operators*, namely with *propagation rules* able to transfer at the global level, and back, the local knowledge. In the example, certain global conclusions, e.g. about

how harmful a certain area is to the robots, could be reached by applying suitable deduction rules to the local information provided by single robots (robot groups).

A third—related—group of foundational models deals with planning and control issues, namely choosing a possible behavior in a non- or indeterministic situation. Apart from the reachability (resp. non-reachability) of desired (resp. undesired) goals, selection criteria might involve priorities (e.g. for restricting complex behaviors), optimization mechanisms and assumptions about competitive behavior (e.g. Nash equilibria). In the example of the robot swarm, the exploration of the terrain could be guided by a Markov decision process that determines the most favorable solution path for exploring the area.

4.5 Performance Models

With respect to performance, the ensembles envisioned in the ASCENS project leave ample room for optimization at various levels. Whenever there is a decision to be made, either at design time, or at runtime, performance is often an important factor. With robot swarms, the algorithms used by the controller may change depending on the environment and internal resources. For example, an ensemble of robots may choose to establish different communication topologies, in order to save power, maximize throughput, or minimize latency. Or they can pool computational power in order to execute a distributed algorithm (e.g. to find a path through a complex terrain of which each robot only sees a part), which might require assigning various tasks to individual robots depending on their available resources, and then combine the results.

To assist in design and development of self-aware systems, we strive to enable modeling of performance critical behavior, both at design time and run-time. At design time, quantitative models will be used for early identification of potential performance problems and to provide an estimate of the resources required for operating within given quality-of-service thresholds. The role of a model at design time is invaluable, because analyses may be conducted even when the actual system is not yet ready to run. Quantitative models will be also employed at run-time, to support autonomy and awareness in ensembles and individual components. Here, the role of the model will be to guide potential adaptations, e.g., establishing a new communication topology, or switching to approximate, but less power demanding algorithms.

Typical techniques for performance modeling and prediction in computer systems include stochastic Petri nets, stochastic process algebras, ordinary or layered queuing networks, and other techniques. These techniques share the problem of state explosion, whereby the size of the state space of the model grows very rapidly with the number of components. In the scenarios envisaged by the ASCENS project, with ensembles consisting of large numbers of components, computational feasibility of a model needs to be taken into account, especially if a model is expected to guide decisions at runtime. A promising approach is to utilize approximate analytical techniques that scale efficiently with population sizes, especially those concerned with continuous-state approximations of (discrete-state) stochastic models. In systems composed of large number of interacting agents, each agent is only characterized by a relatively simple behavior expressed by transitions between states of a small local state space. Crucially, the complexity of this representation is independent of the number of agents, and depends only on the size of the local state space.

Although in principle the existing methods for scalable quantitative analysis appear to be suitable for the purposes of ASCENS, the specific nature of SCs and SCEs presents challenges which prevent simple and direct application of these techniques. The contribution will therefore be in addressing the shortcomings of the existing methods with respect to hierarchical systems and by developing methods for scalable quantitative performance analysis suitable for large-scale systems.

Besides issues with the scalability of performance analysis methods with respect to performance model state space, an important aspect associated with software intensive systems is scalability with

respect to creation of models used for performance analysis and prediction. These were usually created manually by experts not participating on the actual development, which becomes infeasible with increasing size and complexity of a software system, and calls for as much automation in creating performance models as possible. With the spread of model driven software development, it has become possible to reuse information from the system design phase to derive performance models, e.g. using an architectural model of a system enriched with performance-relevant information. With this in mind, we aim to keep correspondence between related models and allow creating models without duplicating information.

5 Validation and Verification: Formal Methods

The complexity and size of ensembles implies that it is difficult to assure that required properties hold and that the system does not exhibit undesirable or dangerous behaviors. Formal methods can help with the validation of system characteristics, but applying them to large, adaptive systems remains a challenging proposition. Research of formal methods in ASCENS has the main goal of addressing some of the obstacles faced in the validation and verification of ensembles.

5.1 From A Posteriori Verification to Constructivity

A big difference between Computer Engineering and more mature disciplines based on Physics, e.g., Electrical Engineering, is the importance of verification for achieving correctness. These disciplines have developed theories guaranteeing the correctness and predictability of artifacts *by construction*. For instance, the application of Kirchhoff's laws allows engineers to build circuits that meet given properties.

Our vision is to investigate links between compositional verification for specific properties and results allowing constructivity. Currently, there exists in Computer Science an important body of constructivity results about architectures and distributed algorithms on which we will build to achieve the goals of ASCENS.

1. We investigate theories and methods for building faithful models of complex SCEs as the composition of heterogeneous SCs, e.g., mixed software/hardware systems. This is a central problem for ensuring correct interoperation, and meaningful refinement and integration of heterogeneous viewpoints. Heterogeneity has three fundamental sources which appear when composing SCs with different (a) execution models, e.g., synchronous and asynchronous execution, (b) interaction mechanisms such as locks, monitors, function calls, and message passing, and (c) granularity of execution, e.g., hardware and software [HS07].

   We need to move from composition frameworks based on the use of a single low-level parallel composition operator, e.g., automata-based composition, to a unified composition paradigm encompassing architectural features such as protocols, schedulers, and buses.

2. In contrast to existing approaches, we investigate two independent directions of compositionality techniques for high-level composition operators and specific classes of properties:

   - One direction is studying techniques for specific classes of properties. For instance, finding compositional verification rules guaranteeing deadlock-freedom or mutual exclusion instead of investigating rules for safety properties in general.
   - The other direction is studying techniques for particular architectures. Architectures characterize the way interactions among SCs are organized. Compositional verification rules

should be applied to high-level coordination mechanisms used at the architecture level of SCEs, without translating them into a low-level automata-based composition.

We expect that the results thus obtained will allow us to identify "verifiability" conditions (i.e., conditions under which verification of a particular property and/or class of SCEs becomes scalable). This is similar to finding conditions for making SCEs testable, adaptable, etc. In this manner, compositionality rules can be turned into correct-by-construction techniques.

Recent results implemented in the D-Finder tool [BBSN08, BBNS09] provide an illustration of these ideas. D-Finder uses heuristics for proving compositionally global deadlock-freedom of SCs, from the deadlock-freedom of its sub-components. Benchmarks published in [BBNS09] show that such a specialization for deadlock-freedom, combined with compositionality techniques, leads to significantly better performance than is possible with general-purpose monolithic verification tools.

A posteriori verification is not the only way to guarantee correctness. System designers develop complex SCEs by carefully applying architectural principles that are operationally relevant and technically successful. Verification should advantageously take into account architectures and their features. There is a large space to be explored, between full constructivity and a posteriori verification. This vision can contribute to bridging the gap between Formal Methods and the body of constructivity results in Computer Science.

Today's autonomous systems provide more coverage for hardware failures than software failures. If they cannot represent and reason about software failures, they are doomed to blind spots and will have limited autonomy. Our goal in ASCENS is to enhance software functionality within the scope of recovery and to develop ensembles that repair their own software.

Many engineering disciplines use monitoring as a major design principle to increase the quality, robustness, and confidence in the correctness of their products. In aircraft engineering, for example, sophisticated automatic controllers are typically monitored to ensure that their predicted state stays within a "stability envelope", from where the system can be safely controlled in a timely manner using slower but better understood and safer techniques. We believe that monitoring can also provide a strong foundation for increasing the quality, robustness, and confidence in the correctness of complex software systems. Monitoring consists of collecting observations from a system's execution and of analyzing these observations to reach some conclusion, for instance, to measure performance or to check conformance to a given specification. We are working on combining monitoring techniques with runtime verification in order to bridge testing and formal verification. Specific areas will include:

1. *Run-time Monitoring*: We study runtime monitoring in the general context of real-time systems and mainly in autonomous systems, i.e., systems whose timing characteristics (e.g., delays between inputs and output) are crucial to the observer. Dealing with such systems requires, on one hand, high-level specification languages which are able to express timing constraints, and on the other hand, techniques to synthesize monitors from such specifications. Members of ASCENS, from Verimag, have been among the first to study monitor synthesis and formal run-time monitoring in the context of real-time systems, for timed automata [BBKT05].

2. *Predictive Analysis*: Being able to detect a specification violation at runtime is very useful in order to have the ability to predict a violation even though it did not occur in the particular observed execution trace. Such a prediction would tell that the violation was close to happen and it could occur under a different execution speed or synchronization of threads, an invaluable piece of information in debugging/testing of concurrent systems.

5.2 Properties of Implementations

While models are extremely important to both understand the system under consideration and to observe and prove its properties, it is the implementation that is at the end executed and whose properties really matter. At this point, it is necessary both (1) to establish and be able to verify correspondence between the models and the implementation and (2) to avoid implementation-specific issues not covered by the models, usually low-level properties such as absence of null-pointer dereferences. While a skeleton of implementation can be partially generated thus providing the correspondence by the correct-by-construction principle, after an evolutionary change of either the models or the implementation their correspondence need to be re-established; otherwise, the properties of the models might be not present in the implementation.

The first aforementioned point will be realized by comparing the model of a system in the SCEL language with its implementation in the C language. This will be possible by extending the GMC model checker by a module being able to process models, to create corresponding state spaces, and, finally, to traverse the state spaces of both the models and the implementation. During state space traversal, GMC will check the properties specified inside the models (e.g., operations and communication sequences) as well as absence of low-level errors (null-pointer dereferences). In the case of an inconsistency or another problem, the developer of the system will be informed in the form of an error trace or another suitable way about the issue.

6 Engineering Ensembles

The three ASCENS case studies present a unique opportunity: we can analyze the successes and failures encountered in the case studies and extract a catalog of challenges, best practices and patterns for engineering ensembles—a catalog derived from experience in three very different application areas. This will present the main contribution of ASCENS to the discipline of ensemble engineering. In addition, a number of tools is being developed by the ASCENS partners; in order to make them more useful for software developers we will integrate them into a development environment.

6.1 Best Practices and Patterns

Designers of a robot swarm like the one mentioned in the introduction are faced with many difficult trade-offs and design decisions: Should they build a swarm consisting of many simple robots or should they rather opt for fewer, more powerful ones? Should the swarm be homogeneous or should it contain robots with specialized capabilities? What kind of knowledge do the robots need? How much knowledge do robots share, how do they assess the quality of the knowledge they acquire from sensors and other robots, and how should robots deal with contradictory information? Should robots have simple, predictable behaviors or more complex ones that have possibly greater potential for adaptation but also for unexpected failures? Should formal methods be used in the development process, and if so, which properties should be validated?

This is just a small selection of the high-level design decisions that have to be taken; during the course of the development innumerable other competing solutions for particular problems, at various levels of detail, have to be evaluated. This will always remain a challenging task that requires experience and domain knowledge on the part of the designer. But best practices can at least help designers to ask the right question, to consider all problems that might arise, and to evaluate the various trade-offs involved in different solutions as objectively as possible.

In the development of traditional software, and in particular in the area of distributed systems, patterns [Fow96, GHJV95] have proven to be a valuable contribution. In general terms, an analysis

or design pattern is a reusable solution to a development problem that specifies the compromises required by the solution as well as its influence on other, related development problems. Pattern libraries provide a uniform vocabulary that simplifies the discussion of design choices, and they are repositories of proven solutions to common design problems.

In ASCENS we want to expand the pattern-based approach to include patterns for key features and mechanisms of SCs and SCEs (*adaptation, awareness, knowledge*, and *emergence*) at different levels of abstraction. An example is [ZBC+11] which includes patterns that help designers to move from "black-box" descriptions (what adaptation, awareness, knowledge and emergence should achieve) to "white-box" solutions (how adaptation, awareness, knowledge and emergence can be realized). In the long term our goal is to provide a semi-formal language for our patterns that allows better integration of the pattern catalog into the ASCENS software development environment. A formal representation of patterns might even enable SCEs to reason about, e.g., structural patterns at run time, and hence use the pattern catalog to autonomously adapt the internal structure of the ensemble.

6.2 Tool Support

Developing SCEs requires dealing with multiple languages, platforms, and tools, which will be provided by the partners or developed within the scope of this project. Regarding the tools, some will be freely available and others might be commercial tools. However, all tools for addressing SCE concerns from both academia and industry can be integrated and used in the Service Development Environment (SDE).

The SDE is a tooling platform which was developed within the scope of the SENSORIA project [MR10, Ser]. It will be used and extended in the ASCENS project to support the engineering of service component ensembles. The core of the SDE allows for a straightforward integration of tools as well as the creation and use of tool chains built as orchestration of tools. Creating a new service as an orchestration of existing services is possible using either a textual, JavaScript-based approach or a graphical workflow approach. The SDE itself is based on a Service-Oriented Architecture, implemented as an Eclipse platform [Ecl11] and its underlying, service-oriented OSGi [OSG08] framework is used.

The key intention is that ASCENS developers integrate all kind of tools for the development of SCEs and for their analysis at design-time or run-time. For example, a tool chain could be defined in the SDE consisting of a modeling tool for the specification of the swarm of robots described in the introduction, a model-checker for validating the specification, and a software for the simulation of the swarm behavior. As the result provided by the modeling tool might not be appropriate as input for the robot swarm simulator, the SDE encourages the definition and use of automated model transformations which translate e.g. between high-level models and formal specifications.

7 Conclusions

The ASCENS takes a "full-stack" approach to the problem of engineering ensembles: we address solutions ranging from high-level modeling and knowledge representation to executable models and run-time monitoring. All languages developed as part of ASCENS are based on solid foundational models and supported by powerful formal methods and tools.

By its very nature, building ensembles will remain a challenging task. Our hope and expectation is that the foundations, tools, languages and engineering methods developed in the ASCENS project will enable developers to build ensembles that are more adaptive, reliable and usable than many of today's systems.

References

[BBBS08] Ananda Basu, Philippe Bidinger, Marius Bozga, and Joseph Sifakis. Distributed semantics and implementation for systems with interaction and priority. In *FORTE*, pages 116–133, 2008.

[BBKT05] Saddek Bensalem, Marius Bozga, Moez Krichen, and Stavros Tripakis. Testing conformance of real-time applications by automatic generation of observers. *Electr. Notes Theor. Comput. Sci.*, 113:23–43, 2005.

[BBNS09] Saddek Bensalem, Marius Bozga, Thanh-Hung Nguyen, and Joseph Sifakis. D-finder: A tool for compositional deadlock detection and verification. In Ahmed Bouajjani and Oded Maler, editors, *CAV*, volume 5643 of *Lecture Notes in Computer Science*, pages 614–619. Springer, 2009.

[BBSN08] Saddek Bensalem, Marius Bozga, Joseph Sifakis, and Thanh-Hung Nguyen. Compositional verification for component-based systems and application. In Sung Deok Cha, Jin-Young Choi, Moonzoo Kim, Insup Lee, and Mahesh Viswanathan, editors, *ATVA*, volume 5311 of *Lecture Notes in Computer Science*, pages 64–79. Springer, 2008.

[Bed97] Mark A. Bedau. Weak emergence. In James Tomberlin, editor, *Philosophical Perspectives: Mind, Causation, and World*, volume 11, pages 375–399. Blackwell Publishers, 1997.

[BS08] Simon Bliudze and Joseph Sifakis. *Causal Semantics for the Algebra of Connectors*. Springer-Verlag, Berlin, Heidelberg, 2008.

[CPZ11] Giacomo Cabri, Mariachiara Puviani, and Franco Zambonelli. Towards a Taxonomy of Adaptive Agent-based Collaboration Patterns for Autonomic Service Ensembles. In *Proceedings of the 2011 International Workshop on Adaptive Collaboration at the International Conference on Collaboration Technologies and Systems (AC/CTS 2011), Philadelphia Pennsylvania, USA, May 2011*, 5 2011.

[DAR] DARPA (Defense Advanced Research Projects Agency). Active networks web site. `http://nms.lcs.mit.edu/darpa-activenet/`, last accessed 2011-11-19.

[DS04] M. Dorigo and T. Stützle. *Ant Colony Optimization*. The MIT Press, 2004.

[Ecl11] Eclipse Foundation. The Eclipse Open Source Community and Java IDE. http://www.eclipse.org/, 2011.

[Fow96] Martin Fowler. *Analysis Patterns: Reusable Object Models*. Addison-Wesley Longman, Amsterdam, 1996.

[GHJV95] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns*. Addison-Wesley, Boston, MA, 1995.

[HRW08] Matthias Hölzl, Axel Rauschmayer, and Martin Wirsing. Engineering of software-intensive systems. In Martin Wirsing, Jean-Pierre Banâtre, Matthias Hölzl, and Axel Rauschmayer, editors, *Software-Intensive Systems and New Computing Paradigms*, volume 5380 of *LNCS*, pages 1–44. Springer, 2008.

[HS07] Thomas A. Henzinger and Joseph Sifakis. The discipline of embedded systems design. *IEEE Computer*, 40(10):32–40, 2007.

[HW11] Matthias Hölzl and Martin Wirsing. Towards a system model for ensembles. In Gul Agha, Olivier Danvy, and José Meseguer, editors, *Festschrift in honor of Carolyn Talcott*, volume 7000 of *LNCS*. Springer, 2011.

[Int] InterLink Project. Website. http://interlink.ics.forth.gr/central.aspx, last accessed: 2011-05-10.

[ITE09] ITEA 2 Office Association. ITEA roadmap for software-intensive systems and services. http://www.itea2.org/, February 2009. 3rd edition.

[Mey08] Bernd Meyer. A tale of two wells: Noise-induced adaptiveness in self-organized systems. In *Proceedings of the 2008 Second IEEE International Conference on Self-Adaptive and Self-Organizing Systems*, pages 435–444, Washington, DC, USA, 2008. IEEE Computer Society.

[ML11] M. Massink and D. Latella. Fluid Analysis of Foraging Ants. Extended Abstract. In *On line Proceedings of the 10th Workshop on Process Algebra and Stochastically Timed Activities (PASTA 2011), Ragusa, Italy, Sept. 19-20, 2011*. Springer-Verlag, 2011. (To appear).

[MLBH11] M. Massink, D. Latella, A. Bracciali, and J. Hillston. Modelling Non-linear Crowd Dynamics in Bio-PEPA. In *Fundamental Approaches to Software Engineering (FASE 2011)*, pages 96–110. Springer-Verlag, 2011.

[MR10] Philip Mayer and Ivan Ráth. D7.4.d: Report on the Sensoria Development Environment (SDE), third version. Deliverable for the eu project sensoria, reporting period october 2008 - february 2010, LMU, 2010. http://www.pst.ifi.lmu.de/projekte/Sensoria/del_54/D7.4.d.pdf.

[OSG08] OSGi Alliance. OSGi Specification Release 4. http://www.osgi.org/Specifications/, March 2008.

[RG03] J.E. Rowe and R. Gomez. El botellón: Modeling the movement of crowds in a city. *Complex Systems*, 14:363–370, 2003.

[SBB01] D. J. T. Sumpter, G. B. Blanchard, and D. S. Broomhead. Ants and agents: a process algebra approach to modelling ant colony behaviour. *Bulletin of Mathematical Biology*, 63:951–980, 2001. doi: 10.1006/bulm.2001.0252.

[Ser] Service Development Environment (SDE). Website. http://svn.pst.ifi.lmu.de/trac/sde, last accessed: 2011-08-11.

[Som07] Ian Sommerville. *Software Engineering*. Addison-Wesley, eighth edition edition, 2007.

[Tof90] C. Tofts. The autosynchronisation of *leptothorax acervorem* (fabricius) described in WSCCS. Technical Report ECS-LFCS-90-128, LFCS, University of Edinburgh, 1990.

[ZBC+11] Franco Zambonelli, Nicola Bicocchi, Giacomo Cabri, Letizia Leonardi, and Mariachiara Puviani. On Self-Adaptation, Self-Expression and Self-Awareness for Autonomic Service Component Ensembles. In *Proceedings of the 1st SASO Workshop on Self-Awareness, Ann Arbor, USA, October 2011*, 5 2011.