www.ascens-ist.eu

ASCENS

Autonomic Service-Component Ensembles

D8.1: First Report on WP8

Challenges of Developing SCEs in the Real World

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 (LMU)**

Due date of deliverable: **September 30, 2011**
Actual submission date: **November 15, 2011**
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

The development of ensembles poses many challenges for the developer beyond those encountered for more traditional software systems. To identify them, the ASCENS case studies provided descriptions of scenarios that highlight important difficulties when developing systems in their respective application area. This deliverable summarizes and generalizes some of the challenges that were identified in these scenarios.

We focus on the following areas: The behavior of ensembles, in particular adaptation, awareness and emergence; models of ensembles and their semantics; robust and scalable formal methods; the role of knowledge and its use; and finally the development process and development tools. Where applicable we give pointers to more detailed descriptions in other deliverables.

Contents

1 Introduction 5
1.1 Essential and Accidental Difficulties 5
1.2 Droves versus Societal Ensembles 6
1.3 Relationship to Other Work Packages and Progress 6
1.4 Structure of this Deliverable 7

2 Behavior of SCs and SCEs 7
2.1 Adaptation, Awareness, Self-Awareness 7
2.2 Adaptation Mechanisms 8
2.3 Fault Tolerance and Security 9
2.4 Limiting Emergence 9
2.5 Engineering Emergence: From Global to Local Behavior and Back 10

3 Models of SCs and SCEs 10
3.1 The Role of Models 11
3.2 Semantics of Models 11
3.3 Surface and Deep Models 12

4 Robust and Scalable Formal Methods 12

5 The Role of Knowledge 13
5.1 Knowledge Representation 13
5.2 Necessary Knowledge 14
5.3 Using Knowledge 14

6 Development Process and Tools 15
6.1 Patterns 15
6.2 Improved Tools and Languages 15
6.3 Domain-Specific Development 16

7 Summary and Next Steps 17
7.1 Summary 17
7.2 Next Steps 17

1 Introduction

Designing and implementing large programs, even for a single computer, is an extremely difficult task. Distributed systems compound these problems with their own issues, such as deadlocks, race conditions, or the lack of uniform global time. On top of that, ensembles present additional challenges to the developer, for example massive numbers of nodes, unpredictable behavior of individual nodes, and open environments. It is therefore clear, that current approaches to software engineering—which can barely cope with traditional software systems—are not sufficient for developing ensembles.

The goal of work package 8 is to eventually come up with patterns, best practices and other software-engineering methods to develop, validate and deploy ensembles. As a first step toward this goal we have analyzed challenges posed by ensembles for the engineer, and some of the most promising approaches to address or overcome them. Given that all the complexities of traditional software and systems engineering also apply to ensemble engineering it is clear that a text of manageable size cannot be an encyclopedic treatment of all such challenges. Instead we focus on those issues which we consider to be the most important obstacles preventing us from building dependable and adaptable ensembles cheaply and reliably.

1.1 Essential and Accidental Difficulties

In two influential articles [FPB87, FPB95] F. P. Brooks points out that the difficulty in the development of software has both *essential* and *accidental* factors. To paraphrase, essential difficulties are those inherent in understanding and addressing the problem the software system is meant to solve, while accidental difficulties are those arising from the tools and methods used in the production of the software. In [FPB87] Brooks identifies four main reasons for essential difficulties in the development of software: the *complexity* of the problems that software addresses, the requirement of *conformity* to complex external interfaces and business processes, the *changeability* of software which leads to continuous demands for additional functionality, and the *invisibility* of software artifacts which prevents us from using physical or spatial intuitions. There have been many arguments about the details of Brooks's theses, and the field of software engineering has made significant progress in the time since the papers were written; but the essence of both papers is still relevant today.

Many of the problems facing the designers of an ensemble are essential; there is little hope of remedying them by proposing solutions for accidental difficulties. For example, increasing the autonomy of the individual components in a system also increases the number of ways in which the components interact and therefore the possibility for undesirable interactions; this is an difficulty that cannot be eliminated in models of the ensemble without removing essential characteristics of its behavior. Most ensembles have high essential complexity, which makes them difficult to understand, describe, implement and test.

We agree with Brooks that it is unlikely that there will be any single method for drastically reducing the effort necessary to deal with the essential complexities, but that a combination of different techniques can ameliorate the problems posed by essential complexities. Many of the challenges presented in this deliverable address specific aspects of the question how essential complexities can be made more tractable.

On the other hand, while it is possible to argue that in the field of traditional software engineering the accidental difficulties have been reduced to such a degree that they can be considered mostly insignificant, this is not the case for ensembles. For example, the question how to concisely describe the goals of a service component and their relation to the goals of the whole ensemble is still a largely unsolved, and at least partly accidental problem.

1.2 Droves versus Societal Ensembles

The nodes of ensembles can exhibit many different levels of sophistication—individual service components can range from simple sensors that have a very limited, fixed behavior to humans (possibly operating complex machines) who may or may not possess intimate knowledge about the workings of the whole system and whose goals may only partially agree with the overall goals of the ensemble. There is no standard terminology to distinguish these kinds of system; in the following we will use the terms *drove* for an ensemble that consists only of simple nodes (so-called reactive service components in the terminology of SOTA described in deliverable D7.1) and *society* or *societal ensemble* for an ensemble containing nodes which are self-aware, adaptive and able to exhibit complex, situation-specific behaviors (goal-based service components).

In many cases the nature of the ensemble determines the kinds of nodes that exist in the ensemble: micro-robots in a robot swarm are limited to simple behaviors by their sensors and computing power, therefore a swarm of such micro-robots will be a drove, whereas an ensemble consisting of human drivers operating electric vehicles will necessarily be societal. When the kind of service components to use is not predetermined by external factors, there is often no agreement which level of complexity these components should exhibit: For example, R. Brooks [Bro91b, Bro91a] argues strongly for building robots without symbolic representation or abstract reasoning and instead relying on emergent behaviors of simple reactive agents; or in the terminology introduced above and applied to ensembles, for droves instead of societal ensembles. On the other side of the argument, for example, R. Reiter [Rei01], and H. Levesque and G. Lakemeyer [LL07] argue for cognitive robotics based on symbolic representation. In practice, most ensembles will be somewhere between these extreme positions, with simple reactive nodes as well as knowledge-based, self-aware nodes operating together. Finding the right structure of the ensemble is one of the largest challenges for designers of ensembles.

In the three ASCENS case studies, swarm robotics is necessarily more oriented towards droves (because of hardware limits of the robots) whereas e-mobility is clearly an example for a societal ensemble. In the case of the science-cloud the situation is not overly constrained by the nature of the ensemble and both kinds of solution appear possible.

Many of the challenges described in this deliverable are relevant for all kinds of ensembles but some, e.g., in Sect. 3.3 or Sect. 5, are only applicable to societal ensembles.

1.3 Relationship to Other Work Packages and Progress

Since WP8 is integrative in nature and relies on the results of other work packages, most of the tasks have been scheduled in the Description of Work to start only in months 13 and 19. The main focus in the first reporting period was therefore on task T8.1 (Challenges of Developing SCEs in the Real World). To this end, we created, in cooperation with WP7, scenario descriptions for each of the case studies; initial versions of these scenarios were made available to all project partners in the first three months of the project, and subsequently refined and clarified. The scenarios present challenges that are typical for the domain of the case study in a way that is easy to understand for the other project partners, and they were widely used in the project to enable and focus collaborative research between project partners. This deliverable summarizes the challenges presented in the scenarios in a more systematic form that focuses on those challenges that are common to ensembles in general. The work of WP8 is related to all other technical work packages; pointers to more detailed discussions of individual challenges in other work packages are given throughout this deliverable. The work in WP8 proceeded as planned and all goals were reached in time.

1.4 Structure of this Deliverable

The structure of this deliverable is as follows: In the next two sections we look at behaviors and models for ensembles, respectively. We then focus on scalable formal methods and the possibilities for using knowledge in ensembles. We then proceed to the development processes and tools for ensembles.

2 Behavior of SCs and SCEs

One of the advantages that, for example, an ensemble of swarm robots promises to have over larger, more powerful, monolithic robots is that the ensemble can be more resilient to partial failure and can more easily adapt to new situations. This flexibility, however, comes at the price of possibly unexpected emergent behaviors that are difficult to recognize and control. In this section we will discuss challenges posed by adaptation, self-awareness and emergence.

2.1 Adaptation, Awareness, Self-Awareness

We call *adaptation* the capability of a system to operate in a number of different environments, or to change its behavior according to new requirements. More formally, we assume that we have a system model describing the ensemble (*Sys*), the environment in which it is operating (*Env*), the goal or desired result of the computation (*Res*), and the connection between the ensemble and the environment (*Link*) which may represent, e.g., a network or sensors. These models can be expressed at various levels of abstraction, depending on the details of the situation. For example, *Sys* might be a non-executable specification of the ensemble, or it might be the executable program running on the individual nodes. We assume that we have a semantic consequence relation $\models$ that describes the relationship between the models and the goal. For more details about this system model for ensembles, see [HW11], a more operational approach based on this system model is SOTA as described in deliverable D7.1.

Then we can describe the correctness of the program in its environment as follows:

$$\mathit{Sys}, \mathit{Env}, \mathit{Link} \models \mathit{Res}. \tag{1}$$

A weaker[1], but often useful, formulation is to ensure that the correct result is not incompatible with the program in a given environment, although the correct result need not be implied (i.e., *Sys*, *Env*, *Link* and *Res* are consistent):

$$\mathit{Sys}, \mathit{Env}, \mathit{Link}, \mathit{Res} \not\models \bot. \tag{2}$$

This latter version allows, e.g., reasoning in many cases where some of the data in the models is still unknown. Note that these formalizations are similar to those used in model-based problem solving [Str07].

Given these definitions we say a system *Sys* can adapt to a range of environments $\mathcal{E}$ (given link *Link* and goal *Res*) if for every $\mathit{Env} \in \mathcal{E}$ we have $\mathit{Sys}, \mathit{Env}, \mathit{Link} \models \mathit{Res}$. More generally, we call a set $\mathcal{A}$ of triples $(\mathit{Env}, \mathit{Link}, \mathit{Res})$ an *adaptation domain* and say that *Sys* can adapt to $\mathcal{A}$ if for every $(\mathit{Env}, \mathit{Link}, \mathit{Res}) \in \mathcal{A}$ we have $\mathit{Sys}, \mathit{Env}, \mathit{Link} \models \mathit{Res}$.

In order to define awareness of a part R of the environment *Env*, we assume that the model of the system *Sys* has a component Z that describes (part of) the internal state of *Sys*, that R is equipped with a distance function d and that we have a map $\rhd$ mapping the domain of Z into the domain of R. Then $d(\rhd Z, R)$ can be seen as a measure of the awareness that *Sys* has of R: if the distance is 0, *Sys* has a perfect internal representation of R (when interpreting Z using $\rhd$); increasing distance means diminishing awareness.

[1]We assume that $\mathit{Sys}, \mathit{Env}, \mathit{Link} \not\models \bot$

Self-awareness can be defined in the same way as awareness by simply substituting the ensemble *Sys* for R. To be useful for practical applications the definitions given in this section have to be extended to take, e.g., probabilistic behaviors into account. However, the simple form of the definitions given above is sufficient for the discussion in the following sections.

2.2 Adaptation Mechanisms

The system model described in the previous section allows us to provide a high-level description of adaptation mechanisms and the consequences that different choices entail. A more detailed discussion of adaptation mechanisms and patterns is given in deliverable D7.1.

The simplest adaptation mechanism is to have a fixed, small number of programs, $P_1, \dots P_n$ running on each node of the ensemble, and a program P_o on each node that selects between P_1 and P_n according to predefined criteria, such as sensor input or commands from other nodes. This results in an ensemble of purely reactive service components and essentially corresponds to a subsumption architecture [Bro91a, Bro91b] or the *Strategy* pattern [GHJV95].

The other extreme would be to have explicit representations of environment, system, link and goals on each node and to use declarative reasoning to arrive at the desired behavior. This is, e.g., the approach taken in cognitive robotics [LL07].

An interesting intermediate approach is to have a set of partial models (or programs) M_i^n, $i \in I$ on each node n such that a valid model (or program) for a given situation can be obtained as

$$M_{J^n}^n = \bigoplus_{j \in J^n} M_j^n$$

where the subset $J^n \subseteq I$ of applicable model elements for node n is determined by the local knowledge of the node, $\oplus$ is a suitable combination operator, and a program $P_{J^n}^n$ can be generated from $M_{J^n}^n$ at run time if $M_{J^n}^n$ is not itself executable. This approach can provide more flexibility than a purely reactive program while using fewer resources for inference than a purely declarative solution. If the $M_{J^n}^n$ are programs, this process roughly corresponds to context-oriented programming [CH05, CH07].

All previously described approaches assume that we have a fixed set of models or building blocks for models. Another possibility is to have certain “adaptation operators” that can modify a class of models. These operators might range from well-understood operations, such as linearly interpolating between values computed by numerical models, to operations with unpredictable consequences, such as random permutations of model fragments, i.e., evolutionary algorithms operating on models. Various experiments in using evolutionary approaches to design circuits with interesting characteristic [TLZ99] and facilities for self-repair [GT04] have been reported and the results imply that in cases where we have millions of parallel nodes, which are difficult to exploit for controlled design, the parallel evolution of several alternative solutions might deliver better results than normal design techniques. A. Thompson notes in [Tho02] that this kind evolutionary design is the only feasible approach in situations where neither the forward nor inverse model are tractable; a situation which might appear frequently in ensembles. The papers [Koz95, KKY$^+$00] by J. R. Koza present an optimistic opinion about the feasibility of this approach. In our estimation it is not clear whether the obtained results are representative. Since it is unlikely that the complete software for an ensemble will be derived by evolutionary techniques, it is an interesting challenge to see how such techniques could be integrated into a development process that relies to a large degree on manually developed components.

Apart from the obvious question how to specify and implement, e.g., the model fragments and combination operators, the previous discussion hints at several interesting challenges which are independent of the way in which the various behaviors on individual nodes are generated: changes between $M_{J^n}^n$ and $M_{K^n}^n$ have to happen when, e.g., node n detects a change of the environment from *Env* to

Env$'$ so that it can no longer satisfy its goal *Res*^{n} using $M^n_{J^n}$. In general it is not easy to determine *that* the environment has changed, or, if a change has occurred, what the new environment is, since the nodes do not possess global knowledge of the current system state. Once a change in the environment has been identified, another challenge is to determine whether adaptation is necessary, i.e., whether $M^n_{J^n}, \mathit{Env}', \mathit{Link} \models \mathit{Res}^n$ still holds for the individual node n and its goal *Res*^{n}. Even if this is the case, it is then necessary to check whether $\bigoplus_n M^n_{J^n}, \mathit{Env}', \mathit{Link} \models \mathit{Res}$ still holds for the overall ensemble, and if not which nodes should adapt. For efficiency reasons it will normally not be possible to prove this formula using automated theorem proving, and more efficient ways to test the validity of this formula have to be investigated for various modeling and programming paradigms. One possible approach that can limit the amount of resources needed for reasoning about the system while still allowing the validation of system correctness is the *negotiate-commit-execute* scheme described in more detail in deliverable D2.1.

Reasoning about, or dynamically changing, the model, is only useful if the results are reflected in the behavior of the system, in other words, there has to be a *causal connection* between the internal model and the program. This is addressed in more detail in Sect. 3.1.

2.3 Fault Tolerance and Security

In the previous section we assumed that the ensemble is exactly described by the model *Sys*. In practice, however, because of the large number of nodes that is characteristic of ensembles, some SCs in the ensemble will be inoperational at any point in time. Similarly, networks or sensors are not 100% reliable, and the environment is open and constantly changing. Therefore, ensembles have to be fault tolerant, and also tolerant to deviations in the environment and the links to the environment. This can formally be expressed by stating that instead of ensuring $\mathit{Sys}, \mathit{Env}, \mathit{Link} \models \mathit{Res}$ we have to ensure that $\mathit{Sys}', \mathit{Env}', \mathit{Link}' \models \mathit{Res}$ for all *Sys*$'$, *Env*$'$ and *Link*$'$ close to *Sys*, *Env* and *Link*, respectively.

On the other hand, many patterns for engineering fault-tolerant systems [Han07] are simplified by the structure of SCEs, since, e.g., SCs form natural *Units of Mitigation*, the large number of nodes in most SCEs aids in adding *Redundancy* and the hierarchical structure provides structured means for *Escalation* of recovery measures.

Similar concerns apply to the issue of security as well. Since ensembles generally operate in open environments it is often possible for adversaries to induce Byzantine attacks [LSP82] against the ensemble; this complex failure mode is often not considered in traditional systems engineering [Han07]. For further challenges about security engineering for ensembles we refer to [And08]

2.4 Limiting 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 we are mostly concerned with this latter denotation of emergence.

An important consideration for adaptive systems, in particular when using evolutionary techniques but also in more controlled approaches, is how to verify that the resulting design still satisfies the specification? It is, after all, not difficult to design scenarios where interaction between different components lead to undesirable weakly emergent behavior.

When working with logical models it might, in some cases, be possible to prove that the result of an adaptation is correct, but in general this will be too resource intensive and unpredictable. Furthermore, this approach presupposes that the composition of well-defined local behaviors does not result in a system that exhibits undesirable behaviors on a global level. This is not generally true as, e.g., the game of Life [Wai74] shows, where even very simple local interactions can lead to unpredictable global behavior [Bee04, GC98]. It is therefore unlikely that it will be possible to ensure that all emergent properties are known at system design time; one of the big challenges for a discipline of ensemble engineering will be to find ways that limit emergent phenomena to stay within a certain range of acceptable system behaviors.

The investigation of architectural patterns and invariants might prove fruitful, e.g., by combining redundancy of the evolved components with a (provably correct) controller that disables the outputs of components not fulfilling their specification. This raises a number of interesting questions regarding the adaptivity of the controller, e.g., how does the controller determine whether its specification still fulfills the requirements in the current environment?

2.5 Engineering Emergence: From Global to Local Behavior and Back

In the previous section we addressed the issue of limiting emergence, but often the converse problem is just as relevant: how do we engineer emergence?

When developing software for ensembles is that the goal is often specified for the whole ensemble, but actions can only be performed by individual nodes, based on incomplete and possibly faulty local knowledge, and with limited capacity to influence their environment. What actions should the nodes take to best achieve the system goal? How do we coordinate tasks and goals between the nodes and the ensemble? What happens when the designers change the goal of the ensemble; how is this change communicated to the nodes and what is the appropriate reaction of the nodes? Similarly, how can nodes decide what action to take in order to achieve the overall system goal when they have only incomplete and possibly unreliable information?

These are all important questions for the designer of an ensemble, in particular for droves, but currently no universally applicable method to address them is known. In swarm robotics, models of behaviors occurring in nature, e.g., the flocking behavior of a school of fish or the foraging behavior of ants, have been investigated, and methods for performing task allocation between different members of a robot swarm have been developed. But as yet there is no general way to reliably combine different behavior in order to achieve more complex tasks (e.g., foraging while keeping the robots organized in a swarm), and no systematic method is known for determining the necessary behaviors of individual robots when the desired behavior of the robot swarm is given.

3 Models of SCs and SCEs

One of the most successful strategies to address complexity is to look at the problem from a higher level of abstraction. This trend has been visible throughout the history of computing, where we have moved from assembly language to modern high-level programming and modeling languages. Currently, object-oriented techniques and languages are prevalent in the development of new software systems; UML models [BRJ05] are routinely used, and have mostly replaced other informal modeling notations in mainstream software development. While these techniques have been successful for traditional software development, the characteristics of ensembles present several challenges which are not adequately addressed by current modeling approaches:

- The role of models as they are currently used is not adequate.
- The formalisms used to model systems often do not have a well-defined semantics.
- Models are mostly “shallow”.

3.1 The Role of Models

As mentioned before, one of the main techniques to enable more and more complex systems to be built is to increase the level of abstraction at which we describe and reason about the system. In software-engineering on essential way to achieve this is the use of high-level models instead of low-level code. It therefore seems likely that the development of ensembles will rely on increasingly high-level models of the system.

In current software development practice models are mainly design-time artifacts: most of the time models are only informally related to the actual system since the implementation is done manually; even in model-driven approaches the model is used to generate the system but the executable code has no “back links” to the model from which it was generated, and therefore no access to its model during runtime. This makes it difficult for a component to dynamically check whether it still performs as foreseen by its designers and prevents it from using the information contained in its models to make choices about desired adaptations and trade-offs.

This may be less of a concern for droves, since the assumption there is that individual agents can operate without knowledge about the system. In ensembles that are oriented more toward the societal organization, nodes should be able to exhibit more sophisticated, situation-specific behaviors. For these kinds of systems, the decisions that have to be taken by individual components to deal with unforeseen situations may be so complex that the simplest way to represent them is for the component to have various models—of its environment, of its own possible behaviors and their effect on the ensemble as a whole, and of its goals and requirements—available at run-time. This would allow the ensemble to reason about the trade-offs implied by various possible responses and choose a course of action that maximizes the expected long-term benefit. Even more powerful possibilities would arise if the different nodes in the systems had a shared notion of key concepts so that nodes could communicate the current situation to peers, and each node could choose an appropriate strategy according to its capabilities and the system’s goals.

One of the main challenges in this area is therefore the connection of models to the executing code. To achieve this, code, models, and goals have to be closely integrated: the program has to be able to determine, while it is executing, which models are relevant to the executing code; on the other hand, if new requirements cause a model to be changed, the code implementing this model has to be modified on the fly to satisfy the updated specifications. Irrespective of automated adaptation of programs taking place, this kind of *traceability* [GF96, THA07] between requirements, models, and code, offers great advantages for the development and debugging of systems.

3.2 Semantics of Models

If models are to be more closely integrated into the actual system, they have to have a well-defined semantics that can unambiguously be understood by both designers, customers and automated tools. Unfortunately this is not the case for UML: the semantics of UML is often complex or non-intuitive, and sometimes inconsistent. The UML specification contains many extension points which deliberately leave certain aspects of the language underspecified. An example of a semantic problem in the definition of UML 2.0 is given in [AS06], where the authors argue that the definition of associations has semantic implications which are not represented in the syntax and may easily be misunderstood. The article [CK04] by S. Cook and S. Kent contains a more detailed discussion of problems with the current UML specification, in particular for the generation of executable code from UML models and the definition of domain-specific extensions for UML.

An important challenge is therefore to develop languages that combine simple, well-specified semantics with high expressivity and good usability for developers. Those languages which are used to communicate with stakeholders should also be understandable for domain experts and customers.

When a suitable language is already widely used in a certain domain, mapping it into the modeling language should be semantically straightforward and achievable by automated tools. This is related to research in areas such as model-driven architecture [MKUW04], where domain models are used to generate executable code and have therefore to be equipped with a precise semantics. In ASCENS this challenge is addressed by the SCEL language described in deliverable D1.1, and also by the foundational models developed as part of WP2 and described in D2.2. Currently the focus is on developing precise and expressive semantic models, later work in WP8 will be concerned with building modeling languages on top of these foundations that are easier to use for developers of ensemble.

3.3 Surface and Deep Models

The distinction between "surface systems" and "deep systems" was defined in an article by Peter E. Hart [Har82] as follows:

> By surface systems I mean those having no underlying representation of such fundamental concepts as causality, intent, or basic physical principles; deep systems, by contrast, attempt to represent concepts at this level.

While the distinction between deep systems (also called "causal models" [Dav82]) and surface systems (which have also been called "shallow models," [Str07] "empirical associations," [Dav82] or "compiled-knowledge systems" [CM82] in the literature) is neither objective nor unambiguous, it nevertheless expresses an important difference. Most models that are currently used in software engineering can be classified as surface models, since they express the information required for the software to function but not the reasons why the system behaves the way it does, or what consequences the actions of the system have.

For example, a model of a university management system typically includes associations between the student and university or course, but it does not contain enough information to conclude that being exmatriculated may have serious consequences for a student, or that posting exam questions on the web may not be advisable before the exam takes place (but might be permissible after the exam is over).

This is not problematic when the models are used by developers as an aid in the development process, or when they are used for code generation purposes, and therefore the greater simplicity and lower cost to build surface models is justified. However, it is not clear which level of adaptivity can be achieved with "shallow" systems, and whether there is a class of behaviors that requires "deep" systems.

The main challenges in this area are finding cost-effective ways to build deep models that are suitable for software engineers without advanced degrees in mathematics or logic. How this can be achieved in practice is still largely a question for future research. There may be interesting confluences with research on rationale management [DMMP06] which tries to capture the design rationales of software developers. The issue of deep models is deeply entwined with the topic of knowledge representation which we address in Sect. 5.

4 Robust and Scalable Formal Methods

A semantically well-defined modeling language enables the use of formal methods, such as model checking or theorem proving, during design time and at run time. While formal methods are not yet widely used in industrial software development, considerable progress has been made in developing efficient, automated tools that can be used by developers to increase the quality of software.

More powerful hardware, improvements in implementation technology and theoretical discoveries have led to tools that can solve increasingly large verification and validation problems automatically. As an example, J. Rushby argues in [Rus06] that SMT (satisfiability modulo theories) solvers can automatically decide certain verification problems that previously required manually guided general theorem proving.

Increasing automation of formal validation and verification methods allows novel ways to use them. For example, formal validation techniques might be used during the negotiation pase of a negotiate-commit-execute cycle to check that the solution that the SCs dynamically agreed upon can actually satisfy the security or privacy requirements of the ensemble.

However, one of the main problems for using formal methods in that kind of scenario is that their performance is often unpredictable: often only the most expert users can accurately estimate whether a certain model or a certain set of parameters can be checked given the available resources, and minor changes to a verification problem can dramatically increase the run-time or memory requirements of validation tools.

If formal methods are to play an important part in the run-time environment of ensembles they will have to be more robust, i.e., the run-time has to be able to reliably determine which tools are applicable in a given situation, e.g., by being able to estimate the expected execution time and the quality of the expected results for various tools.

Another related issue that impacts the use of formal methods in the development of ensembles is the well-known problem of state-explosion: often formal methods are only applicable to small problems because the state space for larger problem instances becomes intractably large. This leads to two complementary challenges: (1) developing methods or tools that can better deal with large state spaces, so that the formal model can directly operate on a model of the ensemble, and (2) finding ways to describe the relevant properties of the ensemble with a significantly reduced state space, e.g., by using stochastic approximations of behaviors.

Detailed information about formal methods and their application to ensembles can be found in deliverable D5.1.

5 The Role of Knowledge

Models for ensembles, in particular deep models, are closely connected to the questions of knowledge representation and the use of knowledge. Here some of the challenges facing the designer of an ensemble are: How should knowledge be represented? What knowledge is needed? How can we capture the domain-specific knowledge, if possible in a reusable manner? How can we use this knowledge?

This section introduces some of the challenges; a detailed discussion of the state of the art and the KnowLang language developed as part of the ASCENS project to address issues of knowledge representation is given in deliverable D3.1.

5.1 Knowledge Representation

One of the main challenges in developing knowledge-intensive systems is finding the right formalism for representing the knowledge. On the one hand, restricted representations such as description logics [BCM$^+$07] allow efficient reasoning about the knowledge base, on the other hand they do so by significantly curtailing the expressive power of the language. Therefore most large-scale knowledge representation projects, e.g., Cyc [Cop97] or SUMO [NP01, SUM] use a very expressive language, often first-order logic with second-order extensions, e.g., for reasoning about collections. But even these systems are often not particularly well-suited for representing complex probabilistic dependencies which arise frequently when modeling real-world problems. Therefore, following the influential books [Pea89, Pea00] by Judea Pearl, Bayesian and causal approaches to knowledge representation have become more popular. However these approaches are mostly restricted to propositional representations; it is not clear that extensions to first-order theories such as Markov logic [DL09] are a generally appropriate solution, and they have not yet gained wide acceptance.

What kinds of knowledge-representation formalisms and reasoning mechanisms are appropriate for ensembles, whether multiple knowledge-representation mechanisms can simultaneously be employed and how knowledge bases specified in different formalisms might be integrated are all important questions about knowledge representation for ensembles that are currently active research areas. In ASCENS these questions are addressed in WP3, and in particular the KnowLang language; more details can be found in deliverable D3.1.

5.2 Necessary Knowledge

The open-ended nature of the environment in which an ensemble operates poses a difficult question for the designer: what kind of knowledge is required by the service components so that they can fulfill the goals set by the designer for the ensemble? The answer to this question may vary widely, depending on the range of possible environments, the complexity of the tasks that the ensemble should fulfill, and the solution chosen by the designer. On the one hand of the spectrum are droves, as they are e.g., currently used in swarm robotics: droves use algorithms that achieve the desired result in a large number of environments without any explicit knowledge about the environment or the tasks that should be achieved. On the other end of the spectrum are systems such as Watson [Wat], the computer that successfully competed in the game show “Jeopardy!” and possesses a huge knowledge base of common-sense knowledge.

A large knowledge base and an expressive knowledge representation formalism may enable a system to operate in more varied environments and to find strategies for achieving better outcomes; however the size of the knowledge base and the expressivity of the representation language also greatly influences the complexity of the reasoning task; it may be that the resources of a system are insufficient to reach any decision in a timely manner if its knowledge base is too large and complex.

Current approaches to software and systems engineering don’t take explicit knowledge into account; knowledge-engineering is generally not overly concerned with the trade-offs faced by software developers. Integrating results from these areas and providing guidance about the required knowledge would be an important step in the direction of developing knowledge-rich ensembles.

Another challenge concerns the reuse of knowledge bases and their extension with domain-specific knowledge: Developing a comprehensive knowledge base is prohibitively expensive and time-consuming and therefore not possible for most projects. On the other hand, it is not clear whether reusable “general-purpose” knowledge bases can be developed and successfully employed in the development process. The free availability of ontologies such as SUMO [NP01] and OpenCyc [Ope] may be an important first step in this direction.

5.3 Using Knowledge

When a knowledge base of the ensemble’s domain is available it may be used at design time or at run time, and in various manners: The knowledge may only be used by (passive) constraints that detect violations of specific properties, it may be used for reasoning whether certain (immediate) goals may be achieved or how they may be achieved, and it may be used for planning and determining long-term strategies.

The use of domain knowledge at design time has been explored in generative and product-line-based software development [GS04, CE00], however the focus of these approaches is mainly on generating families of software systems for related problems. Whether these techniques can be modified to achieve adaptation to different environments is still an open question.

Most forms of reasoning are computationally expensive, therefore it is in general not possible to employ reasoning for every task that a SC has to fulfill. Methods for combining knowledge-based reasoning with more efficient and less resource-intensive ways of performing certain tasks have therefore been investigated for a long time, e.g., in the form of agent architectures such as Touring Machines [Fer92] or 3T [BFG$^+$97]. But architectural solutions do not easily address some challenges for using knowledge at run time, such as: (1) finding ways to use the limited computing power available on the SC for those reasoning tasks that result in tangible benefits, (2) increasing the efficiency and predictability of the reasoning mechanisms, and (3) distributing reasoning across the ensemble while still relying on local knowledge and communication only.

6 Development Process and Tools

To cope with the complexities of ensemble engineering, developers should be supported both by engineering methods and by tools. Furthermore, the increased level of domain modeling necessary for developing societal ensembles poses new challenges for domain-specific development. Parts of this sections are based on [HRW08].

6.1 Patterns

Since the publication of [GHJV95], design patterns and pattern languages have become an important tool for software developers. Patterns are well-understood solutions to common design problems that show the trade-offs for employing a particular solution and provide a vocabulary for talking about the possible designs.

Many of the problems facing ensemble developers are addressed by existing patterns, e.g., networked and distributed computing [SSRB00, BHS07], but no such pattern catalogs exist for other domains, e.g., adaptation or systems with massive scale. One of the goals of WP8 is the development of a system of patterns for those areas.

Patterns are informal tools for software designers; the resulting software usually contains no explicit information which patterns were used and which trade-offs influenced the choice of the particular patterns. This limits the applicability of patterns for ensemble development, in particular for the development of societal ensembles. SCs with deep models (see Sec. 3.3) might exploit this knowledge to dynamically select different solutions to design problems, based on their current knowledge of the environment and goals. Since this selection process is similar to the one that the designer has to go through when developing the software it might be useful for the SC to have knowledge of the patterns that were used to develop the software. How to define formalized patterns that can be used to reason at run time while remaining the generality and flexibility that was one of the reasons for the success of patterns is another challenge that will be investigated by WP8 in the coming reporting periods.

6.2 Improved Tools and Languages

Tools are important to suppress or simplify many of the accidental difficulties in the software development process. For large projects they are also instrumental in navigating models and source code. It is likely that the development of improved tools for programming languages such as Java has played an important role in their industry-wide adoption.

To be useful, improvements in languages have to be accompanied by support for the new features in development environments. For example, being able to combine individual viewpoints into a single program can reduce the effort to develop software. However, just defining the necessary language constructs is, in itself, not helpful. It is also necessary to provide corresponding tools that can display individual viewpoints, as well as the result of combining several or all viewpoints of a system, etc., and to integrate the new language constructs into the navigation mechanisms of the development environment, the debugger, etc.

One challenge for tool support is important is understanding and modifying systems after they have adapted: If the system model changes because the system adapts autonomously to different environmental conditions or requirements, the developer has to be able to understand the reasons for the adaptation and the consequences of the change. Furthermore, individual SCs will, in general, adapt in different ways. When updates to these systems are deployed, they should respect these adaptations (and maybe pass the most successful adaptation to other SCs in similar situations).

Another challenge is the seamless integration of formal methods and formalized patterns into development tools. For example, the tool could extract information contained in the patterns chosen for the system, run tools such as model checkers to find potentially unintended situations,e.g., patterns with incompatible preconditions being used together, and inform the developer about these problems.

6.3 Domain-Specific Development

The domain of a software system plays a prominent role in almost all development approaches, e.g., in the form of requirements analysis and system models [Som07]. Some recent development approaches place particular emphasis on the importance of a detailed understanding of the domain [Eva04] or problem contexts [Jac01], but this is mostly done in an informal context.

Several development approaches try to use domain knowledge in a more formal manner. For example, software product lines [PBvdL05], software factories [GS04], and the step-wise refinement approach of AHEAD [RKW04, BSR04] are approaches based on generative programming [CE00] which propose to develop families of related programs. This is achieved by modeling the domain and possible features of programs and by building generators that can create programs with certain combinations of features and non-functional properties from configuration specifications. As already discussed in Sect. 5.3 it is an interesting challenge to see whether these approaches can be generalized to the development of ensembles.

Similarly, domain-specific languages (DSLs) are sometimes claimed to facilitate the development of certain software systems by providing a simple language for specifying the problem. Traditionally, DSLs are categorized into two kinds: *internal* DSLs where the DSL is embedded in a general-purpose programming language, and *external* DSLs where the DSL is a stand-alone implementation [Fow]. Experience shows that while DSLs often have significant advantages there are also some problems: the behavior of DSLs is often specified by reference to the behavior of an implementation, and in particular external DSLs lead to a proliferation of languages that a developer has to understand. Nevertheless, often the benefits of using DSLs outweigh drawbacks. Techniques for precisely specifying DSLs, for easily deriving an implementation from the specification, and for integrating DSLs into development environment and tools for formal methods remain relevant and challenging research topics, although progress has been made with the availability of tools such as Xtext [Xte].

Returning to Section 3.3, an interesting research topic is the development of theories for various application domains that can be used as deep models, or as background knowledge for deep models. These theories could be equipped with (interfaces to) reasoning components for particular aspects of the domain. For example, a theory for university management might define notions such as "university," "student," "lecture," and their relationships. Associated tools might include a constraint solver for creating timetables and room assignments, or a planner for proposing courses that a student should attend.

7 Summary and Next Steps

This section summarizes the challenges identified during the first reporting period and gives an overview of the work planned in the second year of the ASCENS project.

7.1 Summary

At the beginning of the project, development challenges were identified by elaborating scenarios for the ASCENS case studies. The scenarios were widely used to enable collaborations between partners in the project, and many of the challenges are addressed by ongoing or planned research in ASCENS. In this deliverable we have summarized the challenges in a problem-oriented format and focused on those problems that arise in all three case studies. The main areas in which common challenges were identified are the behavior of SCs and SCEs, models of SCs and SCEs, robust and scalable formal methods, the role of knowledge and the development process.

7.2 Next Steps

With the experience from the case studies the main work of WP8 will begin in the second reporting period, as planned in the Description of Work. The main tasks will be:

- The development of a pattern catalog for ensembles in which patterns for addressing some of the challenges presented in this deliverable will be developed. Our goal is to describe the patterns in a more formal way than most common pattern languages so that the patterns can be made available to development tools or even to SCs at run time.
- The establishment of a repository for SCs. In this task we will try to identify commonly useful software components which are used in the case studies, and make them more easily accessible for developers.

References

[And08] Ross J. Anderson. *Security Engineering: A Guide to Building Dependable Distributed Systems*. Wiley, 2008.

[AS06] C. Amelunxen and A. Schürr. Vervollständigung des Constraint-basierten Assoziationskonzeptes von UML 2.0. In H. Mayr and R. Breu, editors, *Proc. of Modellierung 2006*, volume P-82 of *Lecture Notes in Informatics*, pages 163–172, Bonn, 2006. Gesellschaft für Informatik, Gesellschaft für Informatik.

[BCM$^+$07] Franz Baader, Diego Calvanese, Deborah L. McGuinness, Daniele Nardi, and Peter F. Patel-Schneider, editors. *The Description Logic Handbook: Theory, Implementation, and Applications*. Cambridge University Press, 2nd edition edition, 2007.

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

[Bee04] Randall D. Beer. Autopoiesis and cognition in the game of life. *Artif. Life*, 10(3):309–326, 2004.

[BFG$^+$97] R. Peter Bonasso, R. James Firby, Erann Gat, David Kortenkamp, David P. Miller, and Marc G. Slack. Experiences with an architecture for intelligent, reactive agents. *J. Exp. Theor. Artif. Intell.*, 9(2-3):237–256, 1997.

[BHS07] Frank Buschmann, Kevlin Henney, and Douglas C. Schmidt. *A Pattern Language for Distributed Computing*, volume 4 of *Pattern-Oriented Software Architecture*. Wiley, 2007.

[BRJ05] Grady Booch, James Rumbaugh, and Ivar Jacobson. *The Unified Modeling Language User Guide*. Addison-Wesley, second edition edition, 2005.

[Bro91a] Rodney A. Brooks. Intelligence without reason. In *IJCAI*, pages 569–595, 1991.

[Bro91b] Rodney A. Brooks. Intelligence without representation. *Artif. Intell.*, 47(1-3):139–159, 1991.

[BSR04] Don Batory, Jack Sarvela, and Axel Rauschmayer. Scaling step-wise refinement. *ACM Transactions on Software Engineering (TSE)*, 30(6):355–371, June 2004.

[CE00] Krystof Czarnecki and Ulrich Eisenecker. *Generative Programming: Methods, Tools and Applications*. Addison-Wesley Professional, 2000.

[CH05] Pascal Costanza and Robert Hirschfeld. Language constructs for context-oriented programming: an overview of contextl. In *DLS '05: Proceedings of the 2005 symposium on Dynamic languages*, pages 1–10, New York, NY, USA, 2005. ACM.

[CH07] Pascal Costanza and Robert Hirschfeld. Reflective layer activation in contextl. In *SAC '07: Proceedings of the 2007 ACM symposium on Applied computing*, pages 1280–1285, New York, NY, USA, 2007. ACM.

[CK04] Steve Cook and Stuart Kent. The unified modelling language. In *Software Factories* [GS04], pages 611–627.

[CM82] B. Chandrasekaran and Sanjay Mittal. Deep versus compiled knowledge approaches to diagnostic problem-solving. In *AAAI*, pages 349–354, 1982.

[Cop97] J. B. Copeland. Cyc: A case study in ontological engineering. *Electronic Journal of Analytic Philosophy*, 5, 1997.

[Dav82] Randall Davis. Expert systems: Where are we? and where do we go from here? *AI Magazine*, 3(2):3–22, 1982.

[DL09] Pedro Domingos and Daniel Lowd. *Markov Logic: An Interface Layer for Artificial Intelligence*. Synthesis Lectures on Artificial Intelligence and Machine Learning. Morgan & Claypool Publishers, 2009.

[DMMP06] Allen H. Dutoit, Raymond McCall, Ivan Mistrik, and Barbara Paech. *Rationale Management in Software Engineering*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2006.

[Eva04] Eric Evans. *Domain-Driven Design: Tackling Complexity in the Heart of Software*. Addison-Wesley Professional, 2004.

[Fer92] Innes A. Ferguson. Touring machines: Autonomous agents with attitudes. *IEEE Computer*, 25(5):51–55, 1992.

[Fow] Martin Fowler. Domain specific languages. unpublished. `http://martinfowler.com/dslwip/`, last accessed 2008-09-16.

[FPB87] Jr. Frederick P. Brooks. No Silver Bullet Essence and Accidents of Software Engineering. *Computer*, 20(4):10–19, 1987.

[FPB95] Jr. Frederick P. Brooks. The Mythical Man-Month: After 20 Years. *IEEE Softw.*, 12(5):57–60, 1995.

[GC98] Nicholas Mark Gotts and Paul B. Callahan. Emergent structures in sparse fields of conway's "game of life". In *ALIFE: Proceedings of the sixth international conference on Artificial life*, pages 104–113, Cambridge, MA, USA, 1998. MIT Press.

[GF96] O. Gotel and A. Finkelstein. An analysis of the requirements traceability problem. In R. Arnold and S. Bohner, editors, *Software Change Impact Analysis*. IEEE Computer Society Press, 1996.

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

[GS04] Jack Greenfield and Keith Short. *Software Factories: Assembling Applications with Patterns, Frameworks, Models and Tools*. Wiley, September 2004.

[GT04] M. Garvie and A. Thompson. Scrubbing away transients and jiggling around the permanent: Long survival of FPGA systems through evolutionary self-repair. In C. Metra, R. Leveugle, M. Nicolaidis, and J.P. Teixeira, editors, *Proc. 10th IEEE Intl. On-Line Testing Symposium*, volume 2606 of *LNCS*, pages 155–160. IEEE Computer Society, 2004.

[Han07] Robert Hanmer. *Patterns for Fault Tolerant Software*. John Wiley & Sons, 1 edition, October 2007.

[Har82] Peter E. Hart. Directions for ai in the eighties. *SIGART Bull.*, 79:11–16, 1982.

[HRW08] Matthias M. Hölzl, Axel Rauschmayer, and Martin Wirsing. Software engineering for ensembles. In Wirsing et al. [WBHR08], pages 45–63.

[HW11] Matthias 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: Essays Dedicated to Carolyn Talcott on the Occasion of Her 70th Birthday*, volume 7000 of *LNCS*. Springer, 2011.

[Jac01] Michael Jackson. *Problem Frames: Analysing and Structuring Software Development Problems*. Pearson Education, 2001.

[KKY$^+$00] John R. Koza, Martin A. Keane, Jessen Yu, III Forrest H. Bennett, and William Mydlowec. Automatic creation of human-competitive programs and controllers by means of genetic programming. *Genetic Programming and Evolvable Machines*, 1(1-2):121–164, 2000.

[Koz95] John R. Koza. Survey of genetic algorithms and genetic programming. In *In In Proceedings of the Wescon 95 - Conference Record: Microelectronics, Communications Technology, Producing Quality Products, Mobile and Portable Power, Emerging Technologies*, pages 589–594. IEEE Press, 1995.

[LL07] Hector Levesque and Gerhard Lakemeyer. Cognitive robotics. In *Chapter 24 in Handbook of Knowledge Representation*. Elsevier, 2007.

[LSP82] Leslie Lamport, Robert Shostak, and Marshall Pease. The byzantine generals problem. *ACM Trans. Program. Lang. Syst.*, 4:382–401, July 1982.

[MKUW04] Stephen J. Mellor, Scott Kendall, Axel Uhl, and Dirk Weise. *MDA Distilled*. Addison Wesley Longman Publishing Co., Inc., Redwood City, CA, USA, 2004.

[NP01] I. Niles and A. Pease. Towards a standard upper ontology. In *Proceedings of the 2nd International Conference on Formal Ontology in Information Systems (FOIS-2001)*, 2001.

[Ope] OpenCyc. Web site. `http://www.opencyc.org/`, last accessed 2008-09-15.

[PBvdL05] Klaus Pohl, Günter Böckle, and Frank J. van der Linden. *Software Product Line Engineering: Foundations, Principles and Techniques*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2005.

[Pea89] Judea Pearl. *Probabilistic reasoning in intelligent systems - networks of plausible inference*. Morgan Kaufmann series in representation and reasoning. Morgan Kaufmann, 1989.

[Pea00] Judea Pearl. *Causality: Models, Reasoning and Inference*. Cambridge University Press, 2000.

[Rei01] Raymond Reiter. *Knowledge in Action*. MIT Press, 2001.

[RKW04] A. Rauschmayer, A. Knapp, and M. Wirsing. Consistency checking in an infrastructure for large-scale generative programming. In *Proc. 19th IEEE Int. Conf. Automated Software Engineering (ASE)*, pages 238–247. IEEE, September 2004.

[Rus06] John Rushby. Harnessing disruptive innovation in formal verification. In Dang Van Hung and Paritosh Pandya, editors, *Fourth International Conference on Software Engineering and Formal Methods (SEFM)*, pages 21–28, Pune, India, September 2006. IEEE Computer Society.

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

[SSRB00] Douglas C. Schmidt, Michael Stal, Hans Rohnert, and Frank Buschmann. *Patterns for Concurrent and Networked Objects*, volume 2 of *Pattern-Oriented Software Architecture*. Wiley, 2000.

[Str07] Peter Struss. Model-Based Problem Solving. In van Harmelen et al. [vHLP07], pages 395–465.

[SUM] The suggested upper merged ontology web site. `http://www.ontologyportal.org/`, last accessed 2011-10-17.

[THA07] Bedir Tekinerdogan, Christian Hofmann, and Mehmet Aksit. Modeling traceability of concerns in architectural views. In *AOM '07: Proceedings of the 10th international workshop on Aspect-oriented modeling*, pages 49–56, New York, NY, USA, 2007. ACM.

[Tho02] A. Thompson. Notes on design through artificial evolution: Opportunities and algorithms. In I. C. Parmee, editor, *Adaptive computing in design and manufacture V*, pages 17–26. Springer-Verlag, 2002.

[TLZ99] A. Thompson, P. Layzell, and R. S. Zebulum. Explorations in design space: Unconventional electronics design through artificial evolution. *IEEE Trans. Evol. Comp.*, 3(3):167–196, 1999.

[vHLP07] Frank van Harmelen, Vladimir Lifschitz, and Bruce Porter, editors. *Handbook of Knowledge Representation*. Foundations of Artificial Intelligence. Elsevier, 2007.

[Wai74] Robert T. Wainwright. Life is universal! In *WSC '74: Proceedings of the 7th conference on Winter simulation*, pages 449–459, New York, NY, USA, 1974. ACM.

[Wat] Watson web site. `http://www-03.ibm.com/innovation/us/watson/index.html`, last accessed 2011-10-17.

[WBHR08] Martin Wirsing, Jean-Pierre Banâtre, Matthias Hölzl, and Axel Rauschmayer, editors. *Software-Intensive Systems and New Computing Paradigms - Challenges and Visions*, volume 5380 of *LNCS*. Springer, 2008.

[Xte] Xtext web site. `http://www.eclipse.org/Xtext/`, last accessed 2011-11-03.