www.ascens-ist.eu

ASCENS

Autonomic Service-Component Ensembles

TR 12: A Life Cycle for the Development of Autonomic Systems

Grant agreement number: **257414**
Funding Scheme: **FET Proactive**
Project Type: **Integrated Project**
Latest version of Annex I: **Version 2.2 (30.7.2011)**

Author(s): **Nora Koch (Ludwig-Maximilians-Universität München, Germany), Saddek Bensalem (Université Joseph Fourier Grenoble 1, VERIMAG Laboratory, France), Tomáš Bureš (Charles University, Faculty of Mathematics and Physics, Czech Republic), Jacques Combaz (CNRS, VERIMAG Laboratory, France), Rocco De Nicola (IMT Institute for Advanced Studies Lucca, Italy), Matthias Hölzl (Ludwig-Maximilians-Universität München, Germany), Michele Loreti (Università di Firenze, Italy), Petr Tůma (Charles University, Faculty of Mathematics and Physics, Czech Republic), Martin Wirsing (Ludwig-Maximilians-Universität München, Germany) and Franco Zambonelli (niversità di Modena e Reggio Emilia, Italy)**

Date of technical report: **March 28, 2014**
Revision:
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

Abstract

Autonomic systems are a tool to address increasing complexity of software systems. Their key features are so-called self-* properties, such as self-awareness, self-adaptation, self-expression, self-healing and self-management. In this work we focus on a software development life cycle that helps developers to address the issues posed by the diversity of self-* properties and by engineering adaptive behaviours. In contrast to more classical life cycles, in order to guarantee adaptivity, we rely more on the feedback of runtime data to the design phases. We illustrate how the life cycle can be instantiated using specific languages, methods and tools developed within the ASCENS project. By relying on formal approaches we are able to forecast, analyze and verify systems performance against expected behavior.

Contents

1 Introduction

Software is increasingly used to model or control massively distributed and highly dynamic autonomic systems. One of the main challenges for software engineers is then to find reliable methods and tools to build the complex software required by these systems.

The main aim of the ASCENS project[1] is to tackle these issues using an engineering approach based on service components and ensembles which implement self-* features. Ensembles are dynamic groups of components that are formed on demand to fulfill specific goals by taking into account the state of the (changing) environment they are operating in. One distinguishing characteristic of the approach is the use of formal methods to guarantee that the behaviour of the software complies to the specifications.

In this paper we present the Ensemble Development Life Cycle (EDLC) that covers the full design and runtime aspects of autonomic systems. It is a conceptual framework that defines a set of phases and their interplay mainly based on feedback loops as shown in Figure 1. The life cycle comprises a "double-wheel" and two "arrows" between the wheels providing three different feedback loops: (1) at design time, (2) at runtime and (3) between the two of them. Feedback loop (1) enables continuous improvement of models and code due to changing requirements and results of verification or validation. Loop (2) implements self-adaptation based on awareness about the system and its environment. Finally, loop (3) provides the mechanisms to change architectural models and code on the basis of the runtime behaviour of the continuous evolving system.

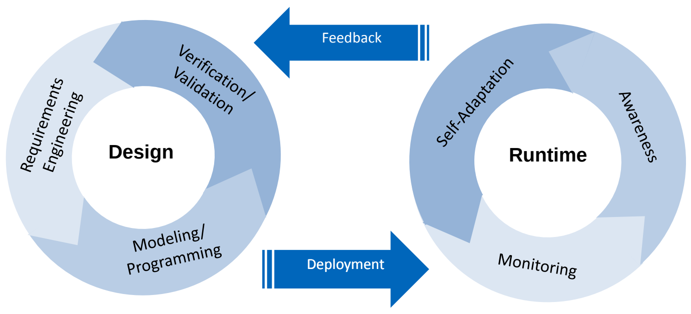

Figure 1: Ensembles Development Life Cycle (EDLC).

We illustrate the EDLC using methods and tools, mostly developed within the ASCENS project. Examples are SOTA for requirements engineering of awareness and adaptive issues, SCEL for modeling and programming, SBIP for verification, SPL for monitoring, Iliad as awareness-engine and JDEECo and JRESP as runtime frameworks. These methods and tools are specifically designed to capture the self-* features of autonomic systems.

The structure of the paper is as follows: Section II provides an overview of the EDLC. Section III focuses on the design phases, i.e. requirements, modeling, programming, verification and validation. Section IV presents tools to support the EDCL runtime phases, i.e. monitoring, awareness and self-adaptation of ensemble-based software systems. In Section V we address the transitions between offline and online engineering activities. To conclude, Section VI relates our work to other relevant software engineering approaches, and Section VII sketches our future plans in the context of EDLC.

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

2 A Software Development Life Cycle

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

Models are usually built on top of the elicited requirements, mainly in following an iterative process, in which also validation and verification in early phases of the development are highly recommended, in order to mitigate the impact of design errors. A relevant issue is then the use of modeling and implementation techniques for adaptive and awareness features. Our aim is to focus on these distinguishing characteristics of autonomic systems along the whole development cycle.

We propose a "double-wheel" life cycle for autonomic systems to sketch the main aspects of the engineering process as shown in Figure 1. The "first wheel" represents the *design* or *offline* phases and the second one represents the *runtime* or *online* phases. Both wheels are connected by the transitions *deployment* and *feedback*.

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

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

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

The process defined by this life cycle can be refined providing details on the involved stakeholders, the actions they perform as well as needed input and the output they produce. A process modeling languages can be used to specify the details. We can use therefore either general workflow-oriented modeling languages such as UML activity diagrams[2] or BPMN[3], or Domain Specific Languages (DSL) such as the OMG standard Software and Systems Process Engineering Metamodel (SPEM)[4] or the Multi-View Process Modeling Language (MV-PML) developed by NASA [BLRV95]. However, describing the refinement and modeling process in detail goes beyond the scope of this overview paper.

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

[2]UML website: http://www.uml.org/

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

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

3 Designing Autonomic Systems

3.1 Requirements Engineering

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

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

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

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

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

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

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

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

3.2 Modeling and Programming

To deal with adaptation and move toward the actual implementation of the self-* properties identified in the previous section, the SCEL language [DLPT13], [DFLP11] has been designed. It brings together programming abstractions to directly address aggregations (how different components interact to form ensembles and systems), behaviors (how components progress) and knowledge manipulation according to specific policies. SCEL specifications consist of cooperating components which, as shown in Figure 2 below, are equipped with an interface, a knowledge repository, a set of policies, and a process.

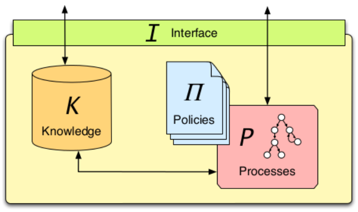

Figure 2: A SCEL component.

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

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

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

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

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

The language is equipped with an operational semantics that permits verification of formal properties of systems. Moreover SCEL program can rely on separate reasoning components that can be invoked when decisions have to be taken. The reasoner is provided with information about the relevant knowledge SCEL program have have access to the programs receive in exchange informed suggestions about how to proceed.

To move toward implementation, jRESP[5] , a JAVA runtime environment has been developed that

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

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

3.3 Verification and Validation

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

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

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

D-Finder is a tool used for the compositional verification of safety properties, that is, it aims at producing proofs stating that ensemble of components cannot reach a predefined set of undesirable states. The method developed combines structural analysis for component behaviours with structural analysis of connectors. The tool implements a compositional method for invariant verification of component-based systems. In contrast to assume-guarantee methods based on assumptions, it relies on interaction invariants to characterize contexts of individual components. These can be computed automatically from component invariants, which play the role of guarantees for individual components. There are two key issues in the application of this method. The first is the choice of component invariants depending on the property to be proved. The second is the computation of the corresponding interaction invariants. Classes of component invariants are used that capture sufficient guarantees for expressing safety properties of the components. The actual computation of invariant does not involve fix-points and avoids state space exploration. D-Finder applies an iterative process for computing progressively stronger invariants. Best precision is achieved when component reachability sets are used as component invariants.

SBIP is an extension of BIP that allows stochastic modeling and statistical verification. On one hand, it relies on BIP expressiveness to handle heterogeneous and complex component-based systems. On the other hand it uses statistical model checking techniques to perform quantitative verification targeting non-functional properties. SBIP relies on a stochastic semantics, which is equivalent to DTMCs and MDPs for the system modeling and probabilistic bounded LTL for property specification. Thanks to great expressive power of BIP, the models used for analysis can be used also to generate concrete implementation to be directly deployed on real platforms. Implementations are guaranteed to be correct that is, they preserve the properties established during the analysis step.

4 Running Autonomic Systems

4.1 Monitoring

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

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

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

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

4.2 Awareness

Awareness comprises the knowledge of the system and its environment as well as the reasoning mechanisms that an ensemble can employ at runtime. We divide the notion of awareness along four dimensions: *scope* (which parts of the system and environment are represented by the awareness mechanism), *breadth* (which properties are part of the awareness model), *depth* (which kinds of questions the awareness mechanism can answer) and *quality* (how well the conclusions the ensemble draws correspond to reality).

To enable problem solving and adaptation in complex domains, *deep awareness mechanisms* may be required. Deep models and reasoners can not only answer questions about the immediately observable state of the system, they also model underlying principles such as causality or physical properties so that they may, e.g., infer consequences of actions or diagnose likely causes of unexpected events.

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

The POEM language [Höl13] enables developers to specify deep logical and stochastic domain models that describe the expected behavior of the system's environment. System behaviors are specified as *partial programs*, i.e., programs in which certain operations are left as non-deterministic choices for the run-time system. A strategy for resolving non-determinism is called a *completion*. Various techniques can be used to build completions: If precise models of the environment are available for certain situations, completions may be inferred by logically or statistically and planning techniques can be used to find a long-term strategy. In cases where models cannot be provided, reinforcement learning techniques can instead be applied, and the ensemble can behave in a more reactive manner.

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

4.3 Self-adaptation

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

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

- *Re-configuration*, aka weak self-adaptation, which implies modifying some of the control parameters of a component/ensemble, and possibly adding new functions/behaviors or modifying some of the existing ones.
- *Self-expression*, aka strong self-adaptation, which implies modifying the very structure of the component or ensemble, and in particular modifying the architecture by which adaptive feedback loops are organized around the component or ensemble.

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

5 Moving between Design and Runtime

The two cycles of EDLC are complemented by transitions from design cycle to runtime cycle and vice versa. These transitions thus correspond to deployment and feedback activities.

5.1 Deployment

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

5.2 Feedback

The feedback transition takes data collected by monitoring a running application back to the design phase to be analyzed and used for improving corresponding components. It connects the runtime monitoring with design. This connection is made possible by employing design methods that keep the traceability of design decisions to code artefacts and knowledge – e.g. Invariant Refinement Method (IRM) [KBP$^+$13], which has been specifically developed for hierarchical design of a service component application). When used in conjunction with IRM, monitoring (a) observes the real functional and non-functional properties of components and situation in components' environment, and (b) provides observed data to the design. At design time these observed data are compared to assumptions and conclusions captured by IRM. If a contradiction is detected, IRM is used to guide a developer to a component or ensemble which has to be adjusted or extended, e.g. to account for an unexpected situation encountered at runtime.

6 Related Work

In the literature we find several approaches for possible architectures or reference models for adaptive and autonomic systems. A well known approach is the MAPE-K architecture introduced by IBM [Cor05] which comprises a control loop of four phases Monitor, Analyse, Plan, Execute. MAPE-K – in contrast to our approach – focus only on the adaptation process at runtime and does not consider the interplay of design and runtime phases. The second research roadmap for self-adaptive systems [dLea11] also suggests a life cycle based on MAPE-K and proposes the use of a process modeling language to describe the self-adaptation workflow and feedback control loops.

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

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

7 Conclusions and Future Work

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

A first proof of concept of the life cycle was performed for the e-mobility domain [BDG+13]. Our future plans include the validation of our engineering approach with more challenging scenarios of different application domains. Future work is also an extension of a standard process modeling language such as SPEM for the visualization of the whole development process.

Acknowledgement

This work has been partially sponsored by the EU project ASCENS, FP7 257414.

References

[AZ12] D. B. Abeywickrama and F. Zambonelli. Model Checking Goal-Oriented Requirements for Self-Adaptive Systems. In *Proceedings of the 19th IEEE International Conference and Workshops on Engineering of Computer-Based Systems*, pages 33–42, April 2012.

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

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

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

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

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

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

[BDG+13] Tomas Bures, Rocco De Nicola, Ilias Gerostathopoulos, Nicklas Hoch, Michal Kit, Nora Koch, Valentina Monreale, Ugo Montanari, Rosario Pugliese, Nikola Serbedzija, Martin Wirsing, and Franco Zambonelli. A Life Cycle for the Development of Autonomic Systems: The e-Mobility Showcase. 2013. submitted.

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

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

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

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

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

[CPZ11] G. Cabri, M. Puviani, and F. Zambonelli. Towards a Taxonomy of Adaptive Agent-Based Collaboration Patterns for Autonomic Service Ensembles. In *Proceedings of the 2011 International Conference on Collaboration Technologies and Systems*, pages 508–515. IEEE, May 2011.

[DFLP11] Rocco De Nicola, Gian Luigi Ferrari, Michele Loreti, and Rosario Pugliese. A Language-Based Approach to Autonomic Computing. In *Formal Methods for Components and Objects, 10th International Symposium, Revised Selected Papers*, pages 25–48, 2011.

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

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

[Höl13] Matthias Hölzl. The POEM Language (Version 2). Technical Report 7, ASCENS, July 2013.

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

[KBP+13] Jaroslav Keznikl, Tomas Bures, Frantisek Plasil, Ilias Gerostathopoulos, Petr Hnetynka, and Nicklas Hoch. Design of Ensemble-Based Component Systems by Invariant Refinement. In *Proceedings of the 16th International ACM Sigsoft symposium on Component-*

based software engineering, CBSE '13, pages 91–100, New York, NY, USA, 2013. ACM.

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

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