Towards a Continuous Model-based Engineering Process for QoS-aware Self-adaptive Systems

Mirko D'Angelo[1], Lorenzo Pagliari[2], Mauro Caporuscio[1], Raffaela Mirandola[3], and Catia Trubiani[2]

[1] Linnaeus University, Växjö (Sweden)
{mirko.dangelo, mauro.caporuscio}@lnu.se
[2] Gran Sasso Science Institute, L'Aquila (Italy)
{lorenzo.pagliari, catia.trubiani}@gssi.it
[3] Politecnico di Milano, Milano (Italy)
raffaela.mirandola@polimi.it

Abstract. Modern information systems connecting software, physical systems, and people, are usually characterized by high dynamism. These dynamics introduce uncertainties, which in turn may harm the quality of service and lead to incomplete, inaccurate, and unreliable results. In this context, self-adaptation is considered as an effective approach for managing run-time uncertainty. However, classical approaches for quality engineering are not suitable to deal with run-time adaptation, as they are mainly used to derive the steady-state solutions of a system at design-time. In this paper, we envision a Continuous Model-based Engineering Process that makes use of architectural analysis in conjunction with experimentation to have a wider understanding of the system under development. These two activities are performed incrementally, and jointly used in a feedback loop to provide insights about the quality of the system-to-be.

1 Introduction

Digitalization of industry, by many considered as the fourth industrial revolution, is changing the competitive landscape in several business domains. The connectivity between software and physical systems opens up for new innovative business or mission critical services responsible for a vast part of the value chain. Indeed, modern information systems (e.g., intelligent transportation systems, smart power grids, network infrastructures and robotics) usually connect software, physical systems, and people [18], who either interact with or are part of the system itself. This means that systems should be designed and developed to explicitly include operational processes and people (e.g., the operators).

Such modern systems are usually characterized by high dynamism, as participating and interacting entities are heterogeneous and autonomous, and unexpected/uncontrolled conditions may arise within the environment. These dynamics introduce uncertainties, which in turn may harm the quality of service and lead to incomplete, inaccurate, and unreliable results [12]. Managing run-time uncertainty is then crucial to operate modern and complex interacting

systems and satisfy their quality requirements. To this end, self-adaptation is considered as an effective approach to manage dynamic interacting systems. In fact, self-adaptive systems are able to adjust their behavior, by hence operating autonomously, in response to their perception of the environment and of the system itself by addressing run-time uncertainties [10].

Engineering modern complex systems exhibiting self-adaptive behavior is challenging, as engineers are required to deal with many different aspects: (*i*) does the type and scale of the system requires for centralized or decentralized adaptation control schemes? (e.g., in terms of coordination among interacting components), (*ii*) does the decisions concerned with the designed behavior introduce overhead? (e.g., in terms of number of exchanged messages), (*iii*) how to validate the quality of different adaptation decisions before putting the system into operation (e.g., in terms of specific quality indicators, such as the system response time, resource utilization, service throughput, etc.)?

To this end, the main research question addressed by this position paper is: *"How to engineer QoS-aware self-adaptive systems by jointly taking into account all the aforementioned challenges?"*.

Classical engineering approaches for quality engineering are not suitable to deal with quality-based adaptation of complex interacting systems, as they mainly employ techniques to derive the steady state solutions of a system at design-time [4].

Therefore, we envision an engineering approach that builds a knowledge-based repository from the continuous and combined use of analytical and experimentation results. This knowledge is built incrementally and used in a continuous model-based engineering process with the aim of providing reasoning support based on the relative costs and benefits for individual design choices.

In our approach the engineer: (*i*) executes the architectural analysis, (*ii*) exploits obtained results for deriving and driving the experimentation phase (e.g., by running only the configurations validated via the architectural analysis), (*iii*) uses the experimentation results to refine/revise the architectural analysis outcome (e.g., some configurations validated by the architectural analysis could be not valid in the experimentation), (*iv*) iterates the process by using the new results as a baseline for the next cycle. Indeed, the overall idea is to combine the benefits of architectural analysis techniques, which are fast but inaccurate for predicting complex states, with experimental results, which on the other hand are accurate but require detailed information on the solution to experiment with.

The paper is organized as follows. Section 2 overviews the overall approach and explains the different phases, namely *design*, *experimentation*, and *feedback*. Therefore, In Section 3 we discuss related work. Finally, Section 4 concludes the paper with hints for future research.

2 The Continuous Model-based Engineering Process

The main idea of the *Continuous Model-based Engineering Process* is to use architectural analysis in conjunction with experimental analysis to have a wider

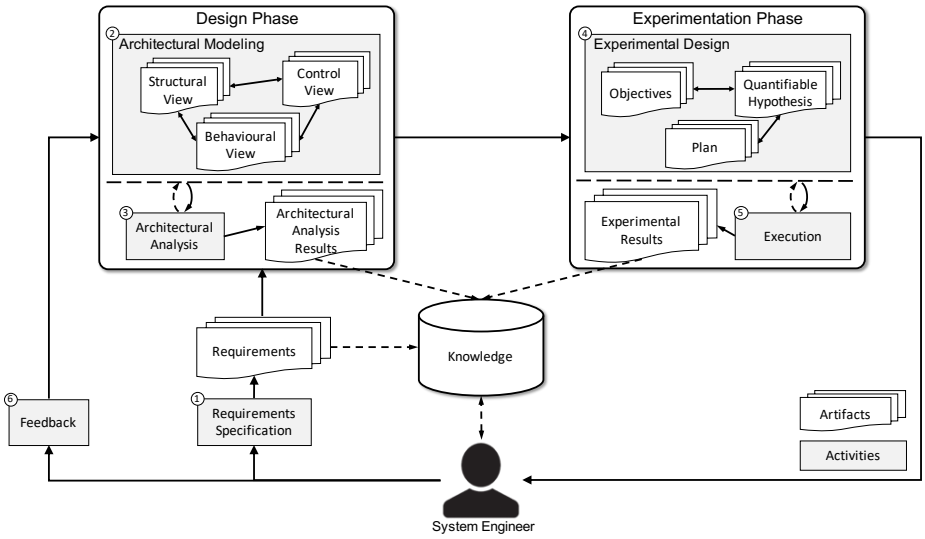

Fig. 1. Continuous Model-based Engineering Process

understanding of the system under development. Indeed, the overall objective is to provide insights about the quality of the system-to-be: the architectural and experimental analyses are performed incrementally and used in the continuous system engineering development process. This supports the system engineers in the process of making informed design decisions on the system under investigation by potentially cutting out or exploring in detail those designed solutions that show either bad or good level of quality.

To this end, the proposed process leverages on the model-based systems engineering paradigm [17], and refers to the systematic use of models as primary artifacts for engineering self-adaptive systems. As shown in Figure 1, the *Continuous Model-based Engineering Process* comprises two different development phases, namely *Design Phase* and *Experimentation Phase*. Specifically: *Requirements Specification* (①) is devoted to eliciting and formally specifying functional and extra-functional requirements for the system, which in turn drive the following phases, as well as the specification and (iterative) evolution of the *Knowledge*. Indeed, the *Knowledge* plays a key role in the envisioned development process as it allows for (iteratively) merging the outcomes of the two phases and providing the System Engineer with a wider view of the system's development. In particular, the knowledge is in charge of linking requirements to design decisions, so that the System Engineer can continuously check and evaluate the different design alternatives with respect to the requirements to be fulfilled. To this end, the knowledge shall be designed as an *Architectural Knowledge* [15] consisting of many different aspects, including (but non limited to): requirements, assumptions, constraints, hypotheses, architecture design decisions, as well as other

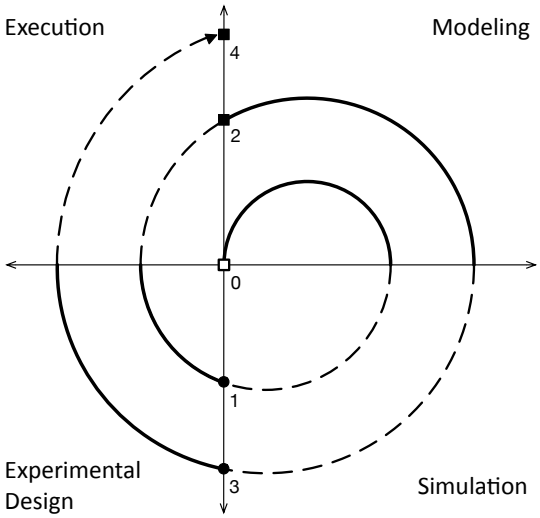

Fig. 2. Knowledge-based spiral model

factors – e.g., the design as built so far, the available technology, best practices, and past experience in the reference domain.

Once the requirements are specified, the *Architectural Modeling* (②) is in charge of modeling the system at a high level of abstraction by means of three different views: (*i*) the *Structural* view defines the system's structure and composition, (*ii*) the *Control* view defines the self-adaptive architecture (e.g., MAPE [20]), and (*iii*) the *Behavioral* view defines the behavior of the system's entities. *Architectural Analysis* (③) performs the model-based analysis of the overall system and checks whether the requirements are satisfied or not. The *Architectural Analysis Results* are then stored into the knowledge for further comparison with the run-time measurements obtained from the experimentation. In fact, jointly considering both design and run-time system's properties allows software engineers to better identify discrepancies between predictions (i.e., architectural analysis results) and measurements (i.e., experimental results) [7].

According to Model-based System Engineering, design-time models might be used to generate (either manually or automatically) the system to be executed. As self-adaptive systems are hard to test in real operational settings, experimentation should be performed in a closed and fully controlled execution environment. To this end, *Experimental Design* (④) aims at deriving, from general objectives, the set of assumptions, constraints, quantifiable hypothesis and phenomenons to be observed and examined during the *Execution* (⑤) of experiments [14]. *Experimental Results* are stored in the knowledge and merged with *Architectural Analysis Results*. The System Engineer can now compare the results and possibly solve observed discrepancies by *feeding back* (⑥) into the

Design Phase. This leads towards a new *Design-Experimentation-Feedback* iteration over the refined/revised system's specification.

Figure 2 shows how the four activities occur in each iteration of the spiral and how they use the knowledge: (*i*) at *Point* 0 the knowledge includes requirements, assumptions, constraints, hypotheses and any other general aspect related to the system context, e.g., best practices, and design patterns; (*ii*) at *Point* 1, at the end of the first Architectural Modeling and Analysis, the knowledge also contains the design decisions taken during the modeling activity and the result of their evaluation against the requirements; (*iii*) all the information contained in the knowledge (e.g., requirements, assumptions, constraints, hypotheses, architectural models, and evaluation results) is used to devise and run the experiments, which in turn enriches the knowledge by providing new results; (*iv*) finally, the knowledge obtained as result of each iteration (*Points* 2 and 4) is used as baseline for the next cycle, where the system specification is further refined/revised and evaluated.

3 Related Work

During the system engineering life-cycle it is fundamental to analyze the behavior of the system under investigation. In particular, it is of key relevance to understand how the designed software alternatives impact the QoS requirements. Two types of analysis can be performed: one which is driven by analytical models, and one emulating the actual system behavior through simulation. The former is typically performed at design time and aims at quantifying as early as possible the QoS characteristics of the systems with analytical and/or QoS-based analysis techniques [5]. The latter is usually used when the resulting system behavior is too complex to be captured by theoretical techniques and more detailed models of the system are introduced to get meaningful QoS-based results.

For the majority of modern systems a satisfactory and omni-comprehensive analysis is highly impractical or even impossible to perform at design time. In fact, in this stage, the system engineer has to verify a complex system with respect to a set of requirements, and there is often no need to consider the precise structure of the system and the details of its elements [16]. When the QoS requirements are not tied to the concrete behavior/execution of the system, high-level QoS models can be selected to preliminarily assess the designed system. Analytical models can be adopted in this phase depending on the specific domain of the system under investigation and the type of QoS requirements to validate. When analyzing QoS characteristics of software systems, many approaches have been proposed to optimize different quality indicators [1]. The main quality attributes that have been evaluated in literature are: performance [19], cost [6], reliability [3], availability [13], but also trade-off analysis among multiple quality attributes has been pursued [11].

When the system exhibits complex behaviors, experimentation-based analysis is usually used to collect results that are otherwise impossible to gather.

Experimentation has been used in software engineering from the 80s [2][14][21] and include (but is not limited to) controlled experiments as well as open-ended exploration. The different methods require for rigorous study design and empirical data analysis to derive insightful and indisputable conclusions from obtained results. In particular, *controlled experiments* relies on: (*i*) the manipulation of the input parameters, (*ii*) the observation of the system's state and output, and (*iii*) an accurate cause-effect analysis.

Experimentation may be considered at different levels. At the system level, experimentation may be used for selecting a specific feature out of a set of alternatives (e.g., A/B testing), whereas at the technical level, experimentation may be used to verify and then optimize a given property (e.g., either functional or non-functional). To this extent, experimentation should be considered as a systemic activity driving the whole development process, from requirement elicitation to verification and validation. This would allow for carefully analyzing the domain hypotheses and assumptions, as well as experimenting the uncertainties. Obtained results would then be turn into knowledge to be incorporated within the decision-making process.

The research landscape on performing design-time and run-time analysis and interpreting the obtained results is less broad, and this is the first factor influencing our research in such a direction. In our previous work [7] we proposed a QoS-based approach that jointly considers design-time and run-time results as complementary aspects of systems. However, the self-adaptation was not considered at all, and there was no interaction by humans in the process. In this position paper we present an approach that deals with these challenges and provides an actual integration of analysis and experimental results in a continuous software engineering life-cycle.

Summarizing, differently from the aforementioned approaches for design-time and/or run-time QoS analysis, our idea is to jointly use architectural analysis and experimental results to provide a continuous knowledge-based engineering process. This way, we aim to support system engineers in the development of QoS-aware self-adaptive systems.

4 Conclusion and Future Work

Modern information systems are characterized by high dynamism, which may introduce uncertainties leading to inaccurate and unreliable results. In this context, employing self-adaptive mechanisms is considered as an effective approach to make a system able to adjust its behavior in response to changes perceived in the environment. However, engineering self-adaptive systems is challenging, as many crosscutting concerns must be jointly accounted during the development process.

In this position paper we envisioned a *Continuous Model-based Engineering Process*, which provides insights about the quality of the system-to-be by incrementally building a knowledge-based repository from the continuous and combined use of analytical and experimental results.

In order to achieve the systematic application of the envioned process, current and future work is towards two different and complementary research directions. On the one hand, we aim at validating and evaluating the process in real-world industrial settings. To this end, we plan to validate the approach applicability in different domains, by employing it in ongoing and future research projects (e.g., Smart Power Grid [8]). We also plan to conduct a controlled experiment with engineers and practitioners from industrial partners, which will design and develop a real fully-featured application in the context of an ongoing research project. Such application will be used to evaluate the efficacy of the process and derive meaningful descriptive statistics.

Further, to fully engineer and (partially) automatize the process, we aim at empowering model-to-code transformations to automatically derive experiments from design artifacts. In the same line of research, we aim at formally specifying the meta-model of the knowledge (e.g., as Architectural Knowledge [9]) to facilitate its instantiation and run-time evolution.

Acknowledgment

This research has received funding from the Swedish Knowledge Foundation, Grants No. 20150088 and No. 20170232. This work has been also partially supported by the PRIN 2017TWRCNB SEDUCE (Designing Spatially Distributed Cyber-Physical Systems under Uncertainty).

References

1. A. Aleti, B. Buhnova, L. Grunske, A. Koziolek, and I. Meedeniya. Software architecture optimization methods: A systematic literature review. *IEEE Trans. Software Eng.*, 39(5):658–683, 2013.
2. V. R. Basili, R. W. Selby, and D. H. Hutchens. Experimentation in software engineering. *IEEE Transactions on Software Engineering*, SE-12(7):733–743, 1986.
3. A. K. Bhunia, L. Sahoo, and D. Roy. Reliability stochastic optimization for a series system with interval component reliability via genetic algorithm. *Applied Mathematics and Computation*, 216(3):929–939, 2010.
4. P. Bocciarelli and A. D'Ambrogio. A model-driven method for enacting the design-time qos analysis of business processes. *Software & Systems Modeling*, 13(2), 2014.
5. R. Calinescu, L. Grunske, M. Z. Kwiatkowska, R. Mirandola, and G. Tamburrelli. Dynamic qos management and optimization in service-based systems. *IEEE Trans. Software Eng.*, 37(3):387–409, 2011.
6. L. Cao, J. Cao, and M. Li. Genetic algorithm utilized in cost-reduction driven web service selection. In *Proceedings of the International Conference on Computational and Information Science*, pages 679–686, 2005.
7. M. Caporuscio, R. Mirandola, and C. Trubiani. Building design-time and run-time knowledge for qos-based component assembly. *Software: Practice and Experience*, 47(12):1905–1922, 2017.
8. M. D'Angelo, A. Napolitano, and M. Caporuscio. Cyphef: a model-driven engineering framework for self-adaptive cyber-physical systems. In *Proceedings of the International Conference on Software Engineering*, 2018.

9. R. C. de Boer, R. Farenhorst, P. Lago, H. van Vliet, V. Clerc, and A. Jansen. Architectural knowledge: Getting to the core. In *Proceedings of the International Conference on Quality of Software Architectures*, pages 197–214, 2007.
10. R. de Lemos et al. *Software Engineering for Self-Adaptive Systems: A Second Research Roadmap*. 2013.
11. A. Doğan and F. Özgüner. Biobjective scheduling algorithms for execution time–reliability trade-off in heterogeneous computing systems. *The Computer Journal*, 48(3):300–314, 2005.
12. D. Garlan. Software engineering in an uncertain world. In *Proceedings of the International Workshop on Future of software engineering research*, 2010.
13. H. Guo, J. Huai, H. Li, T. Deng, Y. Li, and Z. Du. Angel: Optimal configuration for high available service composition. In *Proceedings of the International Conference on Web Services*, pages 280–287, 2007.
14. N. Juristo and A. M. Moreno. *Basics of Software Engineering Experimentation*. Springer Publishing Company, Incorporated, 1st edition, 2010.
15. P. Kruchten, P. Lago, and H. van Vliet. Building up and reasoning about architectural knowledge. In *Proceedings of the International Conference on Quality of Software Architectures*, pages 43–58. Springer-Verlag, 2006.
16. M. Marchiori. Light analysis of complex systems. In *Proceedings of the ACM Symposium on Applied Computing*, pages 18–22, New York, NY, USA, 1998. ACM.
17. S. J. Mellor, A. N. Clark, and T. Futagami. Model-driven development. *IEEE Software*, 20(5):14–18, Sep. 2003.
18. R. Rajkumar, I. Lee, L. Sha, and J. Stankovic. Cyber-physical systems: the next computing revolution. In *Proceedings of the International Conference on Design Automation*, pages 731–736, 2010.
19. V. S. Sharma and P. Jalote. Deploying software components for performance. In *Proceedings of the International Symposium on Component-Based Software Engineering*, pages 32–47, 2008.
20. D. Weyns, B. Schmerl, V. Grassi, S. Malek, R. Mirandola, C. Prehofer, J. Wuttke, J. Andersson, H. Giese, and K. M. Göschka. *On Patterns for Decentralized Control in Self-Adaptive Systems*, pages 76–107. Springer Berlin Heidelberg, Berlin, Heidelberg, 2013.
21. C. Wohlin, P. Runeson, M. Höst, M. C. Ohlsson, and B. Regnell. *Experimentation in Software Engineering*. Springer, 2012.