Design for Sustainability = Runtime Adaptation ∪ Evolution

Danny Weyns, Mauro Caporuscio, Bahtijar Vogel, Arianit Kurti
Deptartment of Computer Science
Linnaeus University
Växjö SE-351 95, Sweden
Contact: danny.weyns@lnu.se; mauro.caporuscio@lnu.se

ABSTRACT

Continuous *change* changes everything; it introduces various uncertainties, which may harm the sustainability of software systems. We argue that integrating runtime adaptation and evolution is crucial for the *sustainability* of software systems. Realising this integration calls for a radical change in the way software is developed and operated. Our position is that we need to *Design for Sustainability*. To that end, we present: (*i*) the AdEpS model (Adaptation and Evolution processes for Sustainability) to handle and mitigate uncertainties by means of integrating runtime adaptation and evolution, and (*ii*) a set of engineering principles to design software systems that facilitate the application of the AdEpS model to build sustainable software.

1. INTRODUCTION

In ecology, *sustainability* refers to the ability of biological systems to remain diverse and productive, i.e., sustainability is the *endurance* of systems and processes. Sustainability is a systemic concept that includes a set of dimensions [3]: *Individual* and *Social sustainability*, *Economic sustainability*, *Environmental sustainability*, and *Technical sustainability*. While *individual*, *social*, *economic*, and *environmental sustainability* are well-established concepts, *technical sustainability*, referring to "the longevity of information, systems, and infrastructure and their adequate evolution with changing surrounding conditions", is an emerging topic, mainly related to the continuous and fast evolution of technologies.

Modern software systems – such as *cyber-physical systems*, *cloud* and *service-oriented systems* – are realised by means of dynamic composition of autonomous and heterogeneous resources that interact with each other to provide users with rich functionalities. Since these systems operate under highly dynamic conditions where both the entities and their interconnections are subject to continuous change, the traditional stability assumptions made on systems' design are no longer valid. The dynamic operating conditions introduce *uncertainty*, which may harm the longevity of the system.

ECSAW '15, September 07 - 11, 2015, Dubrovnik/Cavtat, Croatia

DOI: http://dx.doi.org/10.1145/2797433.2797497 ...$15.00.

Uncertainty can lead to "incomplete, blurred, inaccurate, unreliable, inconclusive, or potentially false" results [30, 28]. Indeed, *uncertainty* spans many different dimensions, such as context, goals, models, functional and quality properties. When *uncertainty* is the rule rather than the exception, managing it becomes a crucial factor for the longevity and thus the sustainability of software systems. In other words, a software system is *sustainable* if it is *resilient to uncertainty*.

Developing sustainable software systems is challenging, as uncertainty permeates virtually all stages of the software system, from *goals elicitation* to *design*, *validation* and in particular *runtime*. This raises a set of groundbreaking challenges that call for radically changing the way software is developed, validated and operated. Software engineering students have been taught for decades the driving principles of *design for reuse*. However, we argue that handling the continuous change of software and realising sustainable systems require a shift in perspective by putting *adaptation* and *evolution* as driving principles in software design. Whereas *adaptation* refers to the ability of mitigating uncertainty in order to keep satisfying the goals, *evolution* refers to the ability of accommodating uncertainty in order to handle goal changes. Phrasing it differently: reuse puts the *stable* parts of software central, but the dominating parts of modern software systems are the *changing* parts; this change requires new engineering principles.

So far, adaptation and evolution have mainly been tackled independently by focusing on runtime and development time issues, respectively. However, the increasing need for *business continuity* requires modern software systems to be continuously available, which blurs the traditional separation between *runtime* and *development time*: uncertainty must be handled when the information becomes available while the system is running. To that end, we argue in this paper for a radically new approach to engineer software that we coin: *Design for Sustainability*. We underpin our argumentation with: (*i*) the AdEpS model (Adaptation and Evolution processes for Sustainability) to handle and mitigate uncertainties by means of integrating runtime adaptation and evolution, and (*ii*) a set of engineering principles to design software systems that facilitate the application of the AdEpS model to build sustainable software.

This paper is structured as follows. Section 2 describes the AdEpS model that integrates runtime adaptation and evolution. In Section 3, we define a set of engineering principles to design software systems that adhere to the AdEpS model. Finally, we reflect on design for sustainability and its realisation, and wrap up with open challenges in Section 4.

2. THE ADEPS MODEL

Almost two decades ago, Oreizy et al. [27] proposed a comprehensive and integrated model that aimed at handling the challenges of adaptation and evolution at run time. Central to this seminal work are two interacting processes, one to handle adaptation, the other one to handle evolution. The Oreizy model takes an architectural perspective on handing change[1] for monitoring, planning, evaluating, coordinating, and implementing reconfigurations. In their FOSE 2007 paper, Kramer and Magee [23] confirmed that software architecture provides a suitable abstraction to deal with change at runtime, which is nowadays widely accepted [7, 9]. Since the introduction of Oreizy's model, research has yielded a set of principle insights in handling change. Figure 1 enhances Oreizy's model, integrating these insights. We call this the AdEpS model (Adaptation and Evolution processes for Sustainability).

- First, we have learned that changing software involves four distinct activities: monitor, analyze, plan, and enact. These four activities are explicitly modeled in the AdEpS model, both for adaptation and evolution.
- Second, over the years, the importance of uncertainty in dealing with change has become manifest. The AdEpS model explicitly handles different types of uncertainty.
- Third, different to the original model of Oreizy, the AdEpS model provides explicit representations of the resources on which the change processes work: (*i*) the architecture description and the system implementation for evolution, and (*ii*) the runtime model and running system for adaptation.

We explain now the two interacting change processes that constitute the AdEpS model in more detail and highlight their interactions.

2.1 Adaptation Management

The aim of *Adaptation Management* (see Figure 1) is to preserve system goals, regarding of dynamics in the system, or the context in which it executes. The central resources of adaptation management are the running system and a runtime architecture model[2]. Adaptation management is usually performed in an automatic way, possibly supported by humans in the loop. Adaptation management monitors the running system and its execution context, updating a runtime architecture model. This model is analysed and when system goals are violated, a plan is selected (or generated) and enacted to adapt the running system accordingly. Such adaptations can range from changing a single parameter value up to changing the architecture configuration. Figure 1 shows different sources of uncertainties involved in adaptation management. The context in which the system executes may be subject of uncertainty; e.g., the availability of resources may dynamically change in ways that are difficult to predict. The runtime architecture model may be subject to uncertainties; e.g., the model may only provide a probability of the response time of a particular component. Monitoring may suffer of uncertainty; e.g., measuring

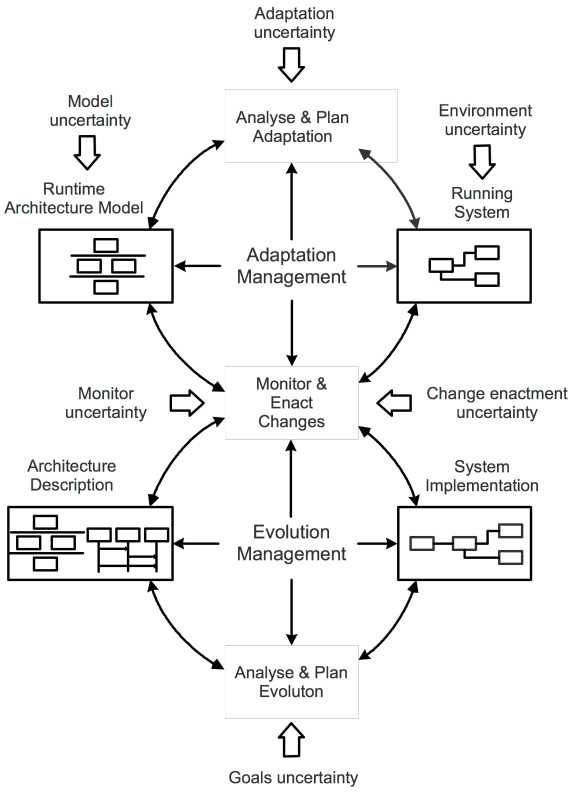

Figure 1: AdEpS model: integrated adaptation and evolution for sustainable software systems

a physical distance may be subject to noise. Change enactment may be subject to uncertainty; e.g., the time to change a service may be different as expected. Finally, there may be uncertainty with respect to the adaptation itself; e.g., the analysis of a problem based on a model abstraction of the real situation may imply inaccuracies.

2.2 Evolution Management

The aim of *Evolution Management* (see Figure 1) is to handle changes of system goals in the context of business continuity; e.g., handle a new user requirement for a 24/7 system. The central resources of evolution management are the system implementation and the architecture description. Evolution management is usually performed by humans, supported by tools. Evolution management can be triggered in two principle ways, see Figure 2.[3] First, adaptation management may trigger the need for evolution, that is, when analysis discovers a problem for which no mitigation plan is available (the problem was not anticipated). This scenario is illustrated in Figure 2(a). When no adaptation plan is available, evolution management will analyse the request. This will result in a plan to update the software architecture and the system implementation. The update is then enacted to the running system and the runtime architecture model. Second, the system goals may evolve due to changing user requirements or other changes in the

[1]With change, we refer both to adaptation and evolution.
[2]The architecture model may include representations of the running system, the context, goals, plans, etc. [34].
[3]We describe the evolution scenarios conceptually. In practice, each step may include different activities, that are performed automatically, semi-automatically, or manually.

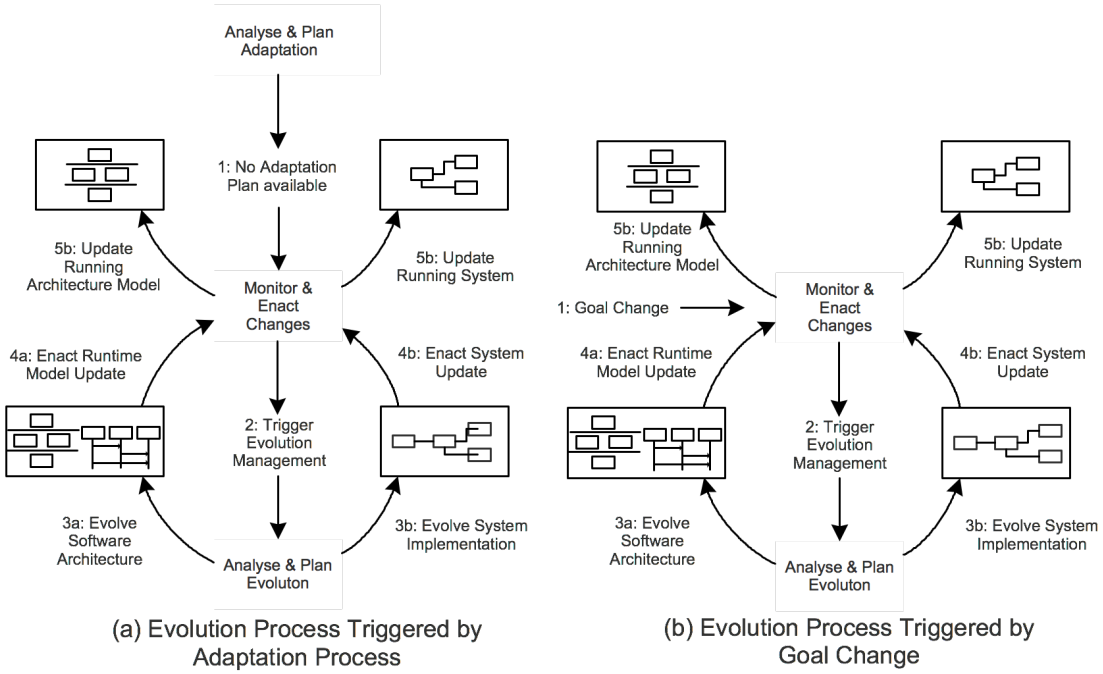

Figure 2: Processes at Work

system or environment. This scenario is illustrated in Figure 2(b). When evolution management gets a request for a goal change, the request will be analysed resulting in an update plan. Subsequently, the architecture description and the system implementation will be updated. The evolution will then trigger an update of the running system and the runtime architecture model. Evolution changes are typically changing or adding new components, integrating platform updates, etc. Evidently, evolution requires synchronization between the adaptation and evolution processes. Evolution may be subject to different uncertainties as shown in Figure 1. Besides uncertainties with respect to monitoring and enacting (similar as explained above), a key uncertainty of evolution management is goals uncertainty; a typical example is a user requirement that was not anticipated before.

From this explanation, it becomes clear that handling runtime adaptation and evolution for sustainability has a pervasive impact on the software. We argue that support for adaptation and evolution should be accounted from the inception of a software system. In the next section, we explain engineering principles that underly design for sustainability.

3. ENGINEERING PRINCIPLES FOR DESIGN FOR SUSTAINABILITY

The AdEpS model defines how a software system can handle change through runtime adaptation and evolution in an integrated way. We shift our focus now to realising software systems that adhere to the AdEpS model, which is a foundational and particularly challenging problem. It requires at least that the system offers runtime support for monitoring its status (and the status of the execution context) and enacting changes (both adaptations and evolution updates). Subsequently, we discuss existing approaches to support runtime adaptation and evolution. We reflect on these approaches and then present a set of engineering principles to support software design for sustainability.

3.1 Existing Approaches to Support Runtime Change

Figure 3 shows the typical progressing levels of maturity to solve problems of software systems over time. Software engineers typically start with solving specific problems in a specific way. When problems recur, the expertise is turned into reusable solutions, for example in the form of frameworks or libraries. In the next stage, engineers abstract from concrete realisations and document design knowledge in the form of architectural approaches to solve the problems, such as tactics, patterns and reference solutions. Then, the knowledge is often consolidated in stable middleware solutions, offering developers programming abstractions and supporting infrastructure. Finally, language support is developed that provides an integrated solution to software developers.

In terms of the Figure 3, researchers and engineers have explored solutions that enable runtime change (adaptation and evolution) of software systems at different levels. We illustrate this with a few examples.

Frameworks – Examples of frameworks are HotSwap [10] that extends the Java VM with the ability to substitute modified code in a running application through the debugger API; Malabara et al. [24] propose an evolution system that extends dynamic class loaders, supporting replacement of class definitions. JDRUMS [1] allows the introduction of new versions of existing Java classes on the fly while preserving the internal state of objects.

FUSION [11] is an advanced framework for engineering self-tuning software systems; one of its particular features is that it supports learning run-time behaviors that were unforeseen at design time.

Patterns, Tactics, Reference Approaches – An exam-

ple of patterns is provided in [36], where the authors document a variety of patterns that support runtime adaptation. One example is the *information sharing pattern* that deals with the problem of adaptation in distributed systems where each sub-system requires information about the state of other sub-systems because a local adaptation may impact these other sub-systems (e.g., on some quality attribute of those operations). The information sharing pattern restricts the interactions between sub-systems to exchange of monitored information. The interactions are typically localised, that is, sub-systems exchange information only with subsystems in their (physical or logical) context.

In [35], the authors present an architectural approach for online updating software product line products. The approach comprises of two complementary parts: (1) an update viewpoint that defines the conventions for constructing and using architecture views to deal with multiple update concerns; and (2) a supporting framework that provides an extensible infrastructure supporting integrators of a SPL. The approach has been empirically validated for live updates of products derived from an industrial SPL for logistic systems [26].

In [29], the authors propose a pattern-oriented development approach, where patterns are considered as the main building blocks of the architecture and changes are applied by means of patterns substitution, i.e., design evolution is identified in terms of replacement of patterns by other patterns. Another interesting approach is proposed in [2], where the authors present the Evolution Style, which defines a family of domain-specific architecture evolution paths that share common properties and satisfy a common set of constraints. The evolution style specifies the set of concepts needed to define and analyse the software architecture evolution: (*i*) the set of operators defining the evolution transitions, (*ii*) the set of evolution path constraints defining whether a path is allowed or not, and (*iii*) the set of evaluation functions used to compare different evolution paths with respect to quality metrics.

Middleware and Component Models – An example of a middleware solution is SOCAM [15] (Service-Oriented Context-Aware Middleware) that supports programmers building and rapid prototyping of context-aware services (i.e., services that use various context data such as environment variables and data about computational elements to adapt themselves to the changing context dynamically and automatically). SOCAM offers support for acquiring, discovering, interpreting and accessing various contexts to build context-aware services.

The Hydrogen project aims at providing mobile devices with a distributed middleware platform for context management and acquisition [20]. Specifically, Hydrogen distributes context servers on devices, which are enabled to share their contexts with other devices in physical proximity. Hydrogen adopts an object-oriented context model that allows the addition of new context types by specializing the generic context superclass. The Contory [31] middleware is specifically designed to accomplish efficient context management on mobile devices. In order to make context management flexible and adaptive, Contory integrates multiple context provisioning strategies, namely internal sensors-based, external infrastructure-based, and distributed provisioning in ad hoc networks.

An interesting example of a component model is Fractal [5] that endows system components with arbitrary reflective capabilities. Components in this model can be endowed with customised introspection and intercession capabilities that allow fine-grained probing and manipulation of their internal structure. Another well-known component model that supports runtime change is OSGi [25]. OSGi defines a dynamic component system for Java, enabling development of applications that are dynamically composed of components. In OSGi components can be remotely installed, started, stopped, updated, and uninstalled on the fly. OSGi is used in popular applications such as Eclipse and Spring.

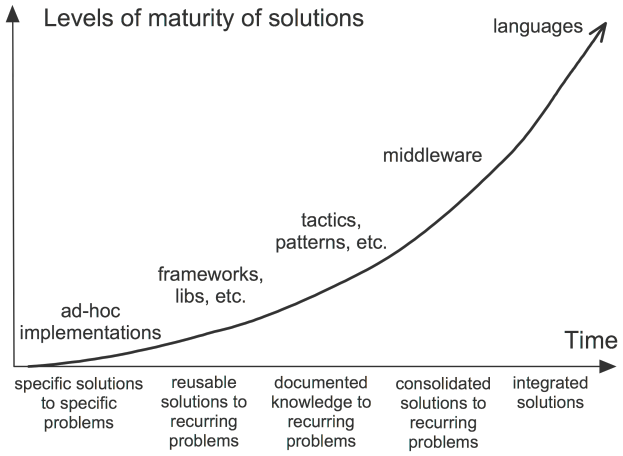

Figure 3: Typical progression of levels of maturity for software system solutions

Language Support – An example of language support is COP [19] (Context-oriented Programming), which treats context explicitly, and provides mechanisms to dynamically adapt behavior in reaction to changes in context at runtime. To that end, COP provides explicit language constructs.

Another famous language example is Erlang [12], a functional programming language and runtime system, designed by Ericsson. Erlang supports runtime evolution at language-level via the "Dynamic Software Updating" mechanism. In particular, Erlang can keep two versions of a module in memory (new and old), and processes can concurrently run code from both. A process will not move into the new version until it makes an external call to its module. Erlang focusses on support for distributed, fault-tolerant, non-stop applications.

3.2 Engineering Principles

The example approaches discussed in the previous section demonstrate that a variety of solutions to support runtime change have been developed over the years. While different solutions provide different levels of maturity, most of them focus on particular facets of "support for runtime change" or apply to particular families of applications. To enable handling runtime adaptation and evolution in a systematic way (i.e., develop software systems that adhere to the AdEpS model), we argue for a paradigm shift in the way we realise software systems. In particular, we argue that sustainability requires that software systems are designed and developed to accommodate runtime adaptation and evolution as a primary concern, rather than an add-on. We call this "De-

sign for Sustainability." We have identified three primary engineering principles that underly design for sustainability: *variability & meta-variability*, *probing*, and *controlled change*. We discuss them now more in detail. For each principle, we outline innovative ideas for realisation at the level of component models and languages.

Design for variability & meta-variability – Variability is interpreted as planned or anticipated change that is pervasive throughout the software lifecycle [13]. The classic way to realise variability is by instrumenting software systems with appropriate variability mechanisms that allow for guided evolution [32]. However, when considering sustainable software systems, variants are needed to realise re-configurations of the system at runtime [4]. The need for variability handling at runtime has been recognised as a key challenge in the domain of Dynamic Software Product Lines [16].

While traditional variability mechanisms may provide pragmatic solutions to this problem, with design for variability we refer to systematic approaches to endow software systems with variability as an integrated property. Such solutions can have different levels of maturity (see Figure 3). For example, a component model may be defined that offers abstractions allowing designers to mark particular elements as variants and add variation points to bind/unbind variants on the fly. Similarly, a programming language may offer language constructs that allow programmers to define parts of the code as variants on the one hand and allow binding/unbinding these variants dynamically at certain points in the code on the other hand.

In addition to support variability as a first-class concern, it is also essential to consider meta-variability, i.e., first-class support for runtime changes of the variability mechanisms themselves. Meta-variability is essential to handle uncertainty with regard to unanticipated changes. So far little research has been done on supporting meta-variability at runtime; an example is [17].

Design for probing – Probing refers to the ability of collecting data about the system's runtime behavior and the context in which it executes. The traditional way of probing of software systems is to instrument the code with facilities to collect data from the running system and its execution context. A classic example at the level of frameworks is a gauge as supported by Rainbow [14]. An example at language level are aspects that allow weaving code into the system to track its behaviour. In [6], such aspect probes are used to continuously evaluate the performance of the system. In general, extracting particular types of information through probing is know to be complex and invasive, for example to support robustness to errors and intrusions.

With design for probing we argue for systematic approaches to endow engineering approaches for software systems with integrated facilities for probing their behaviour, taking into account monitoring uncertainty. For example, a component model (Figure 3) may allow designers to define particular elements of the system as monitor-able. The supporting infrastructure should then allow the adaptation or evolution software (see the AdEpS model) to collect the data at runtime from these monitor-able elements. At a programming language level, a language may offer constructs that allow to declare the runtime behaviour of particular parts of the code as observable. The execution environment should then allow to track the behaviour at runtime and allow client code to get access to the monitored data. First-class support for probing can offer a variety of features to handle monitor uncertainties, such as the ability of monitoring stochastic behaviour, filtering noise, etc.

Design for controlled change – In order to handle change, software systems should be build such that they allow enacting changes. A number of solid solutions exist that provide facilities for changing the running software as we discussed above. Nevertheless, effecting changes is often done in an ad-hoc fashion. In [8], the authors highlight the complexity of enacting change and stress the need for coordinating the adaptation process of software systems. The authors emphasise the importance of the connection between effecting change and dealing with the variability associated with the adaptation process. The authors of [18] argue for the heritage of product lines that provides solid (and practicably) proven engineering foundations, which have been tested in numerous applications. Variability in product lines might relate not only to the variation and evolution of a single system (or its architecture) but to the unifying product line architecture of a myriad of instances of deployed systems, for example in mobile devices and similar deployments, each with different deployment context and configuration options. Such systems have a long life time and need for frequent update over their lifetime, especially when critical functions are increasingly software-defined.

In addition to the means for effecting change, systems should also provide facilities for controlling the change, so that system consistency is preserved both during and after the change enactment. Shutting down and restarting the system is not an option for systems that require business continuity. To that end, to guarantee consistency and avoid disruption of service, the system should be placed in a "quiescence state" – i.e., a state where the system is both passive and has no outstanding transactions which it must accept and service – before the run-time adaptation/evolution is performed [22]. However, although quiescence (or more relaxed, tranquility [33]) is a desirable property, it is difficult to achieve, since quiescent states are not proven to be reachable in bounded time. In addition, changing a software element may require the transfer of state before and after the change and handle interfering behaviour during the change.

Design for controlled change argues for integrated engineering approaches that support enacting change of software systems in a consistent manner, taking into account potential uncertainties of effecting changes. Consistent enacting change of a running software system is in general a very complicated challenge. At the level of a component model, designers may be offered the facilities to define which elements of the system are effect-able and how the change can be enacted. Typically, declared variants are candidates for enactment, while variation points provide the means to enact the change. To guarantee consistency of adaptation, the underlying execution platform needs to support quiescence and state transfer. Supporting consistent change enactment at the programming level is an extremely challenging problem. The key problems are the need for high-level programming abstractions to define: what and how change can be enacted, defining quiescent states, and handling state transfer if necessary. Furthermore, the underlying execution platform should assure that declared variants are replaced in a safe state and state transfers are handled properly when a change is enacted. Last but not least, first-class support for

enacting change should offer support for handling uncertainties of effecting changes. Examples of such features are time windows to effect changes, mechanisms for acknowledging successful (or failed) change actions, etc.

4. CONCLUSIONS AND REFLECTIONS ON DESIGN FOR SUSTAINABILITY

In this position paper, we have argued that integrating runtime adaptation and evolution is crucial for the sustainability of software systems. To handle the continues change of software in a systematic way, we have made a case for a radical new approach to build and operate software that we coined "design for sustainability." This approach comprises two complementary (*i*) the AdEpS model that describes the two integrated processes to handle change, regarding of uncertainties: adaptation management to preserve goals and evolution management to deal with goal changes, and (*ii*) three primary engineering principles to design software systems that adhere to the AdEps model: design for variability & meta-variability, design for probing, and design for controlled change. For each principle, we have indicated innovative ideas for realisation at the level of component models and languages.

Design for sustainability shifts the traditional engineering focus from the stable, reusable parts of software systems to the changing parts that are subject to various uncertainties. Evidently, we have to be careful that we do not swing the pendulum too far. Realising the vision of design for sustainability will be a complex adventure. We conclude this paper with a number of reflections:

- How to realise the seamless integration of runtime adaptation and evolution, that is, how to implement the AdEpS model?
- How to transfer the abstract engineering principles of design for sustainability to a practical realisation?
- What are the implications and tradeoffs of applying the engineering principles of design for sustainability?
- What assurances can be provided for such complex systems that operate in an ocean of uncertainties?

An important challenge along the path of realising the engineering approach proposed in this paper is the need for evaluation, empirical work, and gathering evidence. This applies to the realisation of both the AdEpS model and the engineering principles to design software systems that adhere to the AdEpS model. In our ongoing line of research on ActivFORMS [21], we focus currently on the realisation of the AdEpS model. The approach currently assumes that the underlying managed system provides facilities for monitoring and consistent adaptation. In ActivFORMS, the logic to handle adaptation management is realised by means of formally specified executable models. The models can be verified offline. The models are then connected with the managed system through probes and connectors and are directly executed by a virtual machine. Evolution management is realised on top of adaptation management. Evolution management allows system administrators to track the behaviour of the system, continue verification at runtime, and evolve the adaptation models on the fly when needed. The approach has been evaluated for several application domains with different degrees of criticality, including a robotic system, mobile learning applications, and a service-based medical application to support elderly. We refer the interested reader to the ActivFORMS website.[4]

The challenges to deal with continues change of software are huge; applying ad-hoc engineering practice is risky, for sure in the long term. We have argued that we have to change the way we design and build software if we want to deal with the continuous change and uncertainties of software in a sustainable way. We offer design for sustainability as a proposal to open the discussion.

5. REFERENCES

[1] J. Andersson and T. Ritzau. Dynamic code update in JDRUMS. In *Workshop on Software Engineering for Wearable and Pervasive Computing*, 2000.

[2] J. M. Barnes, D. Garlan, and B. Schmerl. Evolution styles: Foundations and models for software architecture evolution. *Softw. Syst. Model.*, 13(2):649–678, May 2014.

[3] C. Becker, R. Chitchyan, L. Duboc, S. Easterbrook, M. Mahaux, B. Penzenstadler, G. Rodríguez-Navas, C. Salinesi, N. Seyff, C. C. Venters, C. Calero, S. A. Koçak, and S. Betz. The karlskrona manifesto for sustainability design. *CoRR*, abs/1410.6968, 2014.

[4] J. Bosch, R. Capilla, and R. Hilliard. Trends in systems and software variability [guest editors' introduction]. *Software, IEEE*, 32(3):44–51, May 2015.

[5] E. Bruneton, T. Coupaye, M. Leclercq, V. Quéma, and J.-B. Stefani. The fractal component model and its support in java: Experiences with auto-adaptive and reconfigurable systems. *Softw. Pract. Exper.*, 36(11-12):1257–1284, Sept. 2006.

[6] M. Caporuscio, A. D. Marco, and P. Inverardi. Model-based system reconfiguration for dynamic performance management. *Journal of Systems and Software*, 80(4):455 – 473, 2007.

[7] B. H. Cheng, R. Lemos, H. Giese, P. Inverardi, J. Magee, J. Andersson, B. Becker, N. Bencomo, Y. Brun, B. Cukic, G. Marzo Serugendo, S. Dustdar, A. Finkelstein, C. Gacek, K. Geihs, V. Grassi, G. Karsai, H. M. Kienle, J. Kramer, M. Litoiu, S. Malek, R. Mirandola, H. A. Müller, S. Park, M. Shaw, M. Tichy, M. Tivoli, D. Weyns, and J. Whittle. Software engineering for self-adaptive systems: A second research roadmap. In *Software Engineering for Self-Adaptive Systems II*, volume 7475 of *Lecture Notes in Computer Science*. Springer, 2009.

[8] C. E. da Silva and R. de Lemos. Using dynamic workflows for coordinating self-adaptation of software systems. In *2009 ICSE Workshop on Software Engineering for Adaptive and Self-Managing Systems, SEAMS 2009, Vancouver, BC, Canada, May 18-19, 2009*, pages 86–95. IEEE, 2009.

[9] R. de Lemos, H. Giese, H. Muller, M. Shaw, J. Andersson, M. Litoiu, B. Schmerl, G. Tamura, N. Villegas, T. Vogel, D. Weyns, L. Baresi, B. Becker, N. Bencomo, Y. Brun, B. Cukic, R. Desmarais, S. Dustdar, G. Engels, K. Geihs, K. Goschka,

[4] http://homepage.lnu.se/staff/daweaa/ActivFORMS.htm

A. Gorla, V. Grassi, P. Inverardi, G. Karsai, J. Kramer, A. Lopes, J. Magee, S. Malek, S. Mankovskii, R. Mirandola, J. Mylopoulos, O. Nierstrasz, M. Pezze, C. Prehofer, W. SchŁfer, R. Schlichting, D. Smith, J. P. Sousa, L. Tahvildari, K. Wong, and J. Wuttke. Software engineering for self-adaptive systems: A second research roadmap. In *Software Engineering for Self-Adaptive Systems II*, volume 7475 of *Lecture Notes in Computer Science*. Springer Berlin Heidelberg, 2013.

[10] M. Dmitriev. Towards flexible and safe technology for runtime evolution of java language applications. In *Proc. of the Workshop on Engineering Complex Object-Oriented Systems for Evolution*, 2001.

[11] A. Elkhodary, N. Esfahani, and S. Malek. Fusion: A framework for engineering self-tuning self-adaptive software systems. In *Eighteenth ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE '10, 2010.

[12] Ericsson Computer Science Laboratory. Erlang. http://www.erlang.org/.

[13] M. Galster, D. Weyns, D. Tofan, B. Michalik, and P. Avgeriou. Variability in software systems - a systematic literature review. *Software Engineering, IEEE Transactions on*, 40(3):282–306, March 2014.

[14] D. Garlan, S.-W. Cheng, A.-C. Huang, B. Schmerl, and P. Steenkiste. Rainbow: Architecture-based self-adaptation with reusable infrastructure. *Computer*, 37(10):46–54, Oct. 2004.

[15] T. Gu, H. K. Pung, and D. Q. Zhang. A service-oriented middleware for building context-aware services. *Journal of Network and Computer Applications*, 28(1):1 – 18, 2005.

[16] S. Hallsteinsen, M. Hinchey, S. Park, and K. Schmid. Dynamic software product lines. *Computer*, 41(4):93–95, April 2008.

[17] A. Helleboogh, D. Weyns, K. Schmid, T. Holvoet, K. Schelfthout, and W. Van Betsbrugge. Adding variants on-the-fly: Modeling meta-variability in dynamic software product lines. In *Proceedings of the Third International Workshop on Dynamic Software Product Lines (DSPL SPLC 2009)*, pages 18–27, 2009.

[18] M. Hinchey, S. Park, and K. Schmid. Building dynamic software product lines. *Computer*, 45(10):22–26, Oct 2012.

[19] R. Hirschfeld, P. Costanza, and O. Nierstrasz. Context-oriented programming. *Journal of Object Technology*, 7(3):125 – 151, 2008.

[20] T. Hofer, W. Schwinger, M. Pichler, G. Leonhartsberger, J. Altmann, and W. Retschitzegger. Context-awareness on mobile devices - the hydrogen approach. In *HICSS*, 2003.

[21] M. U. Iftikhar and D. Weyns. Activforms: Active formal models for self-adaptation. In *Proceedings of the 9th International Symposium on Software Engineering for Adaptive and Self-Managing Systems*, SEAMS 2014, pages 125–134, New York, NY, USA, 2014. ACM.

[22] J. Kramer and J. Magee. The evolving philosophers problem: Dynamic change management. *IEEE Trans. Softw. Eng.*, 16(11):1293–1306, Nov. 1990.

[23] J. Kramer and J. Magee. Self-managed systems: An architectural challenge. In *2007 Future of Software Engineering*, FOSE '07, pages 259–268, 2007.

[24] S. Malabarba, R. Pandey, J. Gragg, E. Barr, and J. F. Barnes. Runtime support for type-safe dynamic java classes. In *Proc. of the 14th European Conference on Object-Oriented Programming*, 2000.

[25] J. McAffer, P. VanderLei, and S. Archer. *OSGi and Equinox: Creating Highly Modular Java Systems*. Addison Wesley, 2010.

[26] B. Michalik, D. Weyns, N. Boucke, and A. Helleboogh. Supporting online updates of software product lines: A controlled experiment. In *Empirical Software Engineering and Measurement (ESEM)*, 2011.

[27] P. Oreizy, M. M. Golick, R. N. Taylor, D. Heimbigner, G. Johnson, N. Medvidovic, A. Quilici, D. S. Rosenblum, and A. L. Wolf. An architecture-based approach to self-adaptive software. *IEEE Intelligent Systems*, 14(3):54–62, 1999.

[28] D. Perez-Palacin and R. Mirandola. Uncertainties in the modeling of self-adaptive systems: A taxonomy and an example of availability evaluation. In *Proceedings of the 5th ACM/SPEC International Conference on Performance Engineering*, ICPE '14, pages 3–14, 2014.

[29] D. Ram and M. Rajasree. Enabling design evolution in software through pattern oriented approach. In D. Konstantas, M. Léonard, Y. Pigneur, and S. Patel, editors, *Object-Oriented Information Systems*, volume 2817 of *Lecture Notes in Computer Science*, pages 179–190. Springer Berlin Heidelberg, 2003.

[30] J. C. Refsgaard, J. P. van der Sluijs, A. L. Højberg, and P. A. Vanrolleghem. Uncertainty in the environmental modelling process - a framework and guidance. *Environ. Model. Softw.*, 22(11):1543–1556, Nov. 2007.

[31] O. Riva. Contory: A middleware for the provisioning of context information on smart phones. In *Middleware*, pages 219–239, 2006.

[32] M. Svahnberg, J. van Gurp, and J. Bosch. A taxonomy of variability realization techniques: Research articles. *Softw. Pract. Exper.*, 35(8):705–754, July 2005.

[33] Y. Vandewoude, P. Ebraert, Y. Berbers, and T. D'Hondt. Tranquility: A low disruptive alternative to quiescence for ensuring safe dynamic updates. *IEEE Trans. Softw. Eng.*, 33(12):856–868, 2007.

[34] D. Weyns, S. Malek, and J. Andersson. Forms: Unifying reference model for formal specification of distributed self-adaptive systems. *ACM Trans. Auton. Adapt. Syst.*, 7(1):8:1–8:61, May 2012.

[35] D. Weyns, B. Michalik, A. Helleboogh, and N. Boucke. An architectural approach to support online updates of software product lines. In *Proceedings of the 2011 Ninth Working IEEE/IFIP Conference on Software Architecture*, WICSA '11, pages 204–213, Washington, DC, USA, 2011. IEEE Computer Society.

[36] D. Weyns, B. Schmerl, V. Grassi, S. Malek, R. Mirandola, C. Prehofer, J. Wuttke, J. Andersson, H. Giese, and K. Goschka. On patterns for decentralized control in self-adaptive systems. In *Software Engineering for Self-Adaptive Systems II*, volume 7475 of *Lecture Notes in Computer Science*, pages 76–107. Springer, 2013.