Formal Methods and Agile Development: Towards a Happy Marriage

Carlo Ghezzi

1 Introduction

Change is connatural to software: no other human artifact shares this characteristic. Its immaterial nature and lack of physical constraints make it perfectly malleable: any change is in principle possible. Through simple textual operations, software engineers can add, delete, or modify functionalities offered by the software and improve its qualities (e.g., its performance). Technically, both the functional and the non-functional properties may be changed. The ability to provide extremely powerful functionalities in a highly flexible and changeable manner has been the key factor that leads to the current software-dominated world.

Change, however, does not come for free. Despite the apparent simplicity of change operations, it is hard to ensure that changes achieve the desired goals. Change operations operate at a very low level (code level), while the requests for change are dictated by higher level goals, such as adding new functionalities, or speeding up execution, or improving usability. Making sure that changes achieve the new goals, while preserving satisfaction of other unchanged goals, is very often extremely hard. This is the reason why software developers often restrain changes.

Change has been a concern since the 1970s, leading to the research work by Parnas [31–34], Belady and Lehman [6, 27], among others, and the recognition of "software maintenance" as a key concern. Traditional software development processes were mainly structured in a phased, rigidly planned manner, ideally intended to lead to robust processes that would eliminate the need for reconsidering and changing previous design decisions.

C. Ghezzi (✉)
DEIB—Politecnico di Milano, Milano, Italy
e-mail: carlo.ghezzi@polimi.it

© The Author(s) 2018

V. Gruhn, R. Striemer (eds.), *The Essence of Software Engineering*,
https://doi.org/10.1007/978-3-319-73897-0_2

Change, however, turned out to be inevitable in most practical cases. Requirements for a given application are often only vaguely known when a new development starts. They become progressively better known as development proceeds, and feedback information starts flowing from customers and from operation. Knowledge about the operational context in which the software will be embedded is often uncertain at development time. Moreover, even if the requirements and the context are known in the initial stages, they are subject to changes, which may, for example, be due to changes in the wider business context where the application is embedded.

Turbulence in requirements leads to devising alternative software process models, which could naturally accommodate change into the process, to support iterative and incremental development and better align software products to evolving requirements. The term *agile software development* has become popular to collectively characterize a number of industrial efforts aimed at reducing the cost of changes in requirements through multiple short development cycles, rather than long monolithic ones. For a presentation of the different proposals and a discussion of pros and cons, the reader can refer to [30].

Software often supports critical functionalities. Hence it needs to undergo a careful assurance process to assess its *dependability* attributes [1]. The main way agile methods address dependability is through testing. Testing has been effectively integrated into agile development life cycles, leading to what is often called *test-driven development*. Test cases are defined before starting implementation of a new application fragment and stand as a kind of specification for the fragment. They are run on the fragment as soon as it is implemented.

Agile methods focus on adding flexibility and supporting change in the development process. They do so through a feedback loop that involves customers and leads to progressive calibration of requirements. The development cycle, however, runs concurrently with operation. The observations and data gathered on the running software may lead to further changes, which need to be designed, implemented, and then deployed and instantiated. The integration of development and operations in an overall agile framework became known as *DevOps* [5].

The work on agile development has been mainly driven by industry, with little contribution from academic research. By and large academic research has focused more on formal methods to support development and verification through formal models. These efforts lead to approaches that some proposers of agile development even deprecated. The divide between the two worlds has unfortunately widened.

This paper argues that time is now mature for reconciliation of the two worlds. Formal methods developed a stage where they can be effectively incorporated into agile methods to give them rigorous engineering foundations and make them systematic and robust. Rather than being deprecated, they can bring added value and industrial strength to agility. This, however, requires researchers in formal modeling and formal verification to revisit the powerful approaches they developed to make them usable by practitioners and fit the agile world. The purpose of this chapter is exactly to provide arguments in support of this thesis. Arguments will be provided by referring to previously published work. Excerpts from previous publications are

included in this chapter, where appropriate. For complementary discussions of these issues, the reader may refer to [2, 3, 18, 19, 22].

The paper is organized as follows. Section 2 provides a systematic framework to understand and classify software changes. Section 3 focuses on research results that investigated how change can be self-managed by software to achieve increased autonomy. The key concept here is that models and verification should be kept at run time to support continuous run time monitoring and verification as triggers for self-adaptation. Section 4 discusses how formal modeling and verification can be incorporated in the software process to support agile formal verification. This leads to a unified software development and operation approach (DevOps) that is rooted on formal methods. The unified approach is further discussed in Sect. 5. Finally, Sect. 6 concludes the chapter and calls for the further research efforts needed to make the vision of formal methods marrying agile development become true.

2 Understanding Change

Since change is connatural to software, it is important to understand where it comes from so that we can handle it properly. The foundational work by Zave and Jackson [37] on requirements engineering, which sheds the light on software and change, is briefly summarized in Sect. 2.1. This work leads to a useful distinction between *evolution* and *adaptation*, discussed in Sect. 2.2. We will also argue that adaptation can often be anticipated through careful design, and this can lead to development of *self-adaptive* software.

2.1 The Machine and the World

Engineers design machines to perform intended actions in an automated way. Traditional engineers design machines that are powered by chemical, thermal, or electrical means. Likewise, software engineers develop abstract machines, powered by data and algorithms, to satisfy real-world goals.

In their foundational work on requirements engineering, Jackson and Zave observe that software engineers should carefully distinguish between two main concerns: the *world* and the abstract *machine* to be realized by software. The world (also called the *environment*, or the *domain*) is the portion of the real world affected by the machine. The ultimate purpose of the machine is always to be found in the world: the goals to be met and the *requirements* are ultimately dictated by what has to be achieved in the world.

Requirements should be clearly spelled out before developing a machine. They should be expressed in terms of the phenomena that occur in the real world. Some of these phenomena are *shared* with the machine: they are either controlled by the world and observed by the machine—through sensors—or controlled by

the machine and observed by the world—through actuators. The machine is built exactly for the purpose of achieving satisfaction of the requirements in the real world. Requirements *specification* prescribes constraints on *shared phenomena* that must be enforced by the abstract machine to be developed. The goal of the software engineer is to design and implement a machine that is *dependable*, that is, it behaves according to the specification.

To properly understand the requirements and design the software, software engineers need to understand how the affected portion of the world—the embedding environment—behaves (or is expected to behave), because this may affect satisfaction of the requirements. Quoting from [37],

> The primary role of domain knowledge[1] is to bridge the gap between requirements and specifications.

As a simple example, consider the design of a robotized system whose goal is to move boxes from a point A to a point B on a flat surface, assuming that no obstacles are placed between A and B. To satisfy the requirement, the software might instruct an actuator to apply a suitable force in the direction A to B. Kinematics laws express environment knowledge in this example. The relevant law here is that to move the box in a certain direction, we need to apply a force in the same direction, whose magnitude exceeds the friction of the box with the surface. In other terms, there is an *environment property* that ensures satisfaction of the requirement if the software correctly implements the functionality of sending a suitable force command to the actuator (and, of course, assuming that the actuator works properly).

As another example, consider the design of an e-commerce system whose goal is to support user interactions ensuring a given maximum response time. To provide a solution that satisfies the response time requirement, a software engineer needs to make certain *environment assumptions*. For example, she needs to make assumptions about the maximum rate of request submissions from customers. Under a given assumption, she can design a solution that ensures satisfaction of the requirement.

Notice that in the previous discussion we distinguished between environment "properties" and "assumptions." The distinction is crucial, although it is not always obvious. By "property" we mean a statement that cannot be falsified. Typically, it expresses a law of behavior that has been proven by an accepted theory, as in the case of kinematics above. By "assumption" instead we mean a statement that can be falsified. It may express uncertain or changeable knowledge. Very often, it represents partial or uncertain knowledge that we have when the system is being designed, which may only become known when the system will be running. User profiles (like in the previous e-commerce example) are a typical example. They are hard to predict, and they may change over time. Other increasingly common examples of uncertainty about the environment arise in the case of virtualized run time environments (cloud computing, service-oriented computing),

[1] The terms environment and domain are used interchangeably.

where the environment has hard-to-predict effects on non-functional requirements like performance or reliability.

The approach described by Jackson and Zave provides a formal conceptual framework to express in mathematical logic what we described informally so far. Let R be a set of logical statements that formalize the requirements, and let S be a set of logical statements that formalize the machine specification; let EP and EA be sets of logical statements that formalize the environment properties and assumptions, respectively. Assuming that S, EP, and EA are all satisfied (i.e., the software is correct with respect to S and the environment satisfies EP and EA) and consistent with each other, the designer's responsibility is ultimately to ensure the following entailment relation:

$$S, EP, EA \models R \quad (1)$$

Equation (1) formalizes the *main dependability argument* that software engineers need to make as part of assurances for their artifacts.

2.2 Evolution and Adaptation

Hereafter we discuss how changes can be classified and how they may affect the dependability argument described by Eq. (1).

Changes may affect environment assumptions. The environment might behave according to a different set of assumptions—say EA'—which may lead to breaking the dependability argument. As we already mentioned, this is a rather common case, since environment assumptions embody uncertain and changeable knowledge. If this happens, and requirements cannot be changed (e.g., weakened), assurance of the dependability argument requires that S should also change, and hence the software implementation. This kind of software change can be called *adaptation*. Changes may also affect the requirements. As mentioned earlier, requirements are highly volatile in practice, and a change of requirements inevitably leads to a change in the software. This kind of software change can be called *evolution*.

Adaptation can often be self-managed by software. Advances in research in the past decade have shown that if the sources of possible environment changes may be anticipated, one may design the software in a way that it monitors changes, analyzes their potential effects, and self-adapts accordingly, if necessary. The main findings of research on self-adaptive software are reported in Sect. 3.

Not all environment changes can be anticipated. Certain phenomena that may affect requirements satisfaction may be initially overlooked and are discovered only later, thus leading to changes in the software. Expert human inspections are required to discover unanticipated dependencies and to plan redesign activities that may lead to a correct solution. Expert human intervention is also needed to elicit and specify requirements changes and then change the software accordingly.

As we discussed earlier, requirements changes are pervasive, from the initial conception throughout all the software lifetime. The need to structure the software lifetime around the notion of change leads to devising agile methods. In the sequel, we will elaborate the notions of change discussed in this section to show how formal methods can be amalgamated with agile methods to engineer dependable software.

3 Achieving Self-adaptive Software

The goal of making software self-adaptable has been a hot research topic in the last decade and many promising results have been proposed. For a broad view of the area, the reader may refer to the series of SEAMS workshops and symposia[2] and the two Dagstuhl reports [11, 13].

Engineering self-adaptive systems calls for specific new approaches to the development and operation of software that guarantee lifelong requirements fulfillment in the presence of environmental changes. A particularly relevant—and perhaps prevailing—case concerns self-adaptation to keep satisfying *non-functional requirements*, such as reliability, performance, and different kinds of cost-related requirements, such as energy consumption. Non-functional requirements are often quite sensitive to environment changes. For example, response time to queries (performance) may depend on traffic assumptions. Likewise, heavy traffic may cause denial of service and thus affect service reliability.

We outline an approach that can (1) predict possible requirements failures caused by changes occurring in the environment and (2) self-adapt by triggering appropriate countermeasures that dynamically reconfigure the running application to prevent breaking the dependability argument. For a further discussion on self-adaptation to preserve satisfaction of non-functional requirements, the reader can refer to [20].

To be self-adaptive, a software system must be able to (1) detect the relevant changes in the external world in which it operates, (2) reason about its own ability to continue to fulfill the requirements as a consequence of the detected changes, and (3) reconfigure itself to guarantee a seamless adaptation to the new external conditions.

Several promising approaches to software self-adaptation rely on the use of *models at run time* [9, 10]. Past work of the author and co-authors fully embraced this view: models are kept alive at run time and updated automatically as changes are dynamically discovered through monitoring (see [14–16, 20, 21]). Formal models are kept at run time to support automatic detection of possible requirements violations. Different kinds of operational models may be kept, each specialized to detecting specific requirements violations. For example, Markovian models can be used to model performance, reliability, and performance, as discussed in [17]. The model's state space can be systematically and exhaustively explored through *on-line model checking* against a formal description of requirements expressed as

[2] http://www.self-adaptive.org.

logic statements. The outcome of model checking may trigger proper adaptation strategies to steer system reconfigurations and prevent requirements violations. Conceptually, this framework establishes a *feedback control loop* between models and the running system. At run time, monitored environment data are fed back to generate possible model updates, which are in turn analyzed against requirements. Adaptation thus becomes model driven. This approach reflects and formalizes the *autonomic control loop* advocated in [25].

Whenever verification at run time fails, the currently running application because the environment does not behave according to the current assumptions, the application should try to self-adapt. To make the application self-adaptive, different approaches have been proposed. In the increasingly common case where the application is structured as a service-oriented architecture, dynamic binding to external services may try to solve the problem [23]. It is also possible to address the problem by designing the application as a dynamic software product line [24]. In any case, dynamic reconfiguration must occur while the application is running. Several techniques have been devised to support dynamic software reconfigurations in a completely safe, nondisruptive, and efficient way [4, 26, 36].

To make the approach practical, run time verification must be performed efficiently. If verification is performed by model checking, most mainstream techniques and tools cannot be adopted. Existing techniques, in fact, were originally defined to support off-line (development-time) analysis and are not meant for on-line usage, where they need to comply with strict real-time constraints. Run time verification must in fact support timely adaptation, to avoid unacceptable disruption in service provision. To solve this problem, solutions have been developed to bring model checking to run time.

A possible solution is based on the observation that changes often are not radical and have a local scope. This assumption allows model checking to be made *incremental*. For example, [15] shows how models described as discrete time Markov chains can be incrementally checked in a very efficient way against temporal probabilistic requirements. The approach is based on the hypothesis that uncertain and changeable assumptions about the environment can be encoded as model parameters (specifically, as unknown probabilities associated with transitions), which can then be estimated at run time through monitoring.

In conclusion, self-adaptation to changes in the environment can be achieved under the following assumptions:

1. During design, it is possible to identify the possible sources of uncertainty.
2. A parametric model can be produced for the system where parameters encode environmental conditions that may change and become known during operation. This allows computing simple verification conditions that must be evaluated at run time.
3. Environmental data can be collected at run time to provide the actual values of model parameters, which can be fed into the verification conditions.
4. The application is structured in a way that it can be reconfigured dynamically in order to accommodate run time parameter variability.

4 Supporting Dependable Evolution

Agile methods support software development in an iterative and incremental manner to accommodate continuous change. In the initial phases of a new project, requirements must be progressively calibrated and exploration of different design alternatives must be supported. Subsequently, as soon as a version of software is deployed and running, concurrent development activities produce new versions, which may, for example, improve satisfaction of non-functional requirements, meet additional requirements, or deal with environment changes that were not anticipated and not supported by self-adapting policies of the current operational versions.

Current state-of-the-art practices in agile development mainly address the organizational aspects involved in iterative and incremental development. To support product quality assurance, they advocate *test-driven development—TDD*. In essence, TDD prescribes that each product increment should be initially specified by designing the test cases the implementation should run successfully, and then relies on automation of test case execution. Although this emphasis on continuous quality assurance is commendable, more can and should be done to achieve dependability.

Advances in formal modeling and formal verification have led to results that may be incorporated in the practice of software development. To achieve this, however, more software engineering research is needed to align formal methods to the needs of practitioners. To support explorative design, verification should be possible also on incomplete (partial) formal models. It is also necessary to provide techniques to formally verify models in face of their evolution [22, 35].

In an agile approach, software development is structured through frequent iterations: an incomplete specification evolves into a complete one once the unknown or uncertain aspects become known. In addition, parts of the system can be deliberately left incomplete at a given stage, and their completion is postponed to a later stage. Suppose that model checking is used to verify the various iterations. To do so, we need a model-checking procedure that can support both *reusability* and *incrementality* [28]. Reusability matters because changes to a model may have a local impact, and thus redoing the whole verification after any change, as several existing model-checking approaches require, would be very inefficient and would become a bottleneck in practice. Furthermore, often software is designed through an iterative decomposition, to support prioritization of different parts and separate developments. This requires that it should be possible to complete a specification incrementally. It would thus be useful to be able to check if an incomplete specification meets the specified requirements. In the likely case that satisfaction of the global property depends on the missing components, it would be desirable to know under which constraints the missing parts would satisfy it. Verification of these constraints would be later performed by analyzing only the added parts.

To tackle these problems, the incremental approach presented in [29] allows state models to include *black-box* states, that is, states that encapsulate an unspecified behavior, whose design is deferred to a later stage. The verification procedure checks the incomplete design against a formal property in temporal logic, expressing a

global system requirement. The outcome of verification can be *OK*, if the model satisfies the property of *KO* if it does not. The outcome can also be *MAYBE*, if there is a possible refinement of the black box that may lead to a violation. In this last case, the verification procedure also synthesizes a formal property (called *proof obligation*) that expresses a constraint on the unspecified part to be met in order to satisfy the global requirement.

Another relevant line of research has focused on making analysis incremental. Incrementality is a necessary feature to be supported if one wishes to make formal verification practically usable. Since iterative development is based on continuous relatively small changes, the ability to verify if changes keep satisfying requirements is of paramount importance. An incremental verification approach reuses the results of previous analysis to verify an artifact after change, and tries to minimize the portion of new analysis to be performed. It may explain the outcomes of analysis in terms of the changes. It has expected benefits over a non-incremental approach in terms of speed. It also has benefits since it helps focusing analysis on the scope of a change. The principle of incrementality can be applied to verification of any artifact, for example, both models [7] and code [8].

5 Towards a Unified View of Development and Operation

Agility emerged as an important principle in the software industry. Development and operation are viewed as iterative, interacting processes that may lead to quality products that better satisfy customer needs. We argued that software engineering research on formal models and verification, rather than being an obstacle to agility, can help to make a substantial step forward in making agile methods more rigorously founded and ultimately more robust.

By marrying agile and formal methods, we can envision the process shown in Fig. 1. The process is *model and verification driven* and is based on rigorous mathematical foundations. The design phase is an iterative process (1) in which models for the requirements, the environment, and the software are progressively developed and formally verified. Requirements are progressively refined, along with the specification of environment properties and assumptions. The software models designed by engineers are continuously verified against requirements. The models are transformed into an executable implementation, which is then deployed in the target environment. The running application monitors the environment and provides actual data that can be used to update the assumptions on which the current model is based. Verification at run time can lead to self-adaptation. The run time self-adaptive loop (2) may fail and require intervention by the software engineer. This is represented by the feedback loop (3).

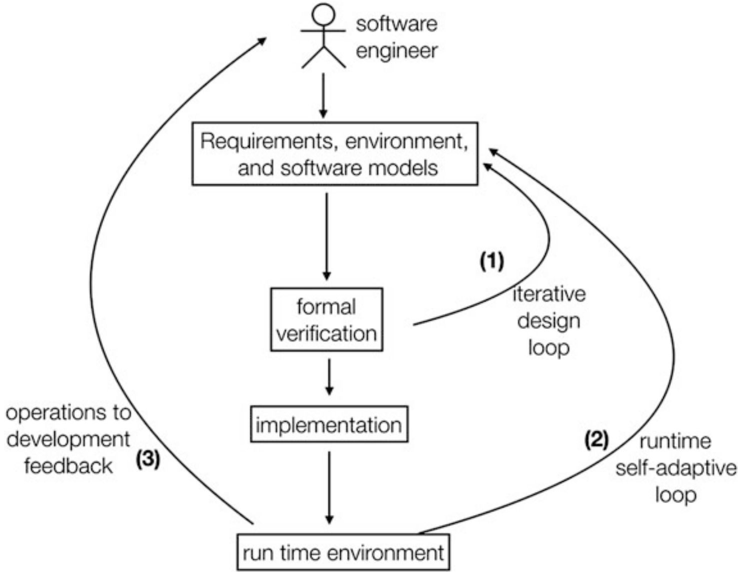

Fig. 1 The development and operation process

6 Concluding Remarks

Progress in software engineering has been remarkable in many directions. Agile methods acknowledged that change has to be handled as a primary concern and recommended iterative processes to cover development and operation in a seamless fashion. Progress has also been remarkable in formal modeling and verification. Software engineers, like engineers in other fields, can now design and analyze models of their artifacts before implementing them. Model-driven engineering has developed automated tools to support derivation of implementations from models [12]. Models and verification can be kept at run time to support self-adaptation.

It is now time to reconcile agility and formal methods. The demand for agility coming from the practitioners' world can be empowered by the rigorous and systematic foundations provided by formal methods. To make this view possible, further research is needed to adapt modeling and verification to fit into the iterative nature of agile processes. Here we reviewed some of the current research efforts moving in this direction, but more remains to be done. This chapter can be viewed as a call for a collective effort that encompasses both researchers and practitioners.

References

1. Avizienis, A., Laprie, J.-C., Randell, B., Landwehr, C.: Basic concepts and taxonomy of dependable and secure computing. IEEE Trans. Dependable Secure Comput. **1**, 11–33 (2004)
2. Baresi, L., Ghezzi, C.: The disappearing boundary between development-time and run-time. In: Proceedings of the FSE/SDP Workshop on Future of Software Engineering Research, FoSER '10, pp. 17–22 (2010)
3. Baresi, L., Di Nitto, E., Ghezzi, C.: Toward open-world software: issue and challenges. IEEE Comput. **39**(12), 36–43 (2006)
4. Baresi, L., Ghezzi, C., Ma, X., Panzica La Manna, V.: Efficient dynamic updates of distributed components through version consistency. IEEE Trans. Softw. Eng. **43**(4), 340–358 (2017)
5. Bass, L., Weber, I., Zhu, L.: DevOps: A Software Architect's Perspective. Addison-Wesley Professional, Reading (2015)
6. Belady, L.A., Lehman, M.M.: A model of large program development. IBM Syst. J. **15**(3), 225–252 (1976)
7. Bianculli, D., Filieri, A., Ghezzi, C., Mandrioli, D.: Incremental syntactic-semantic reliability analysis of evolving structured workflows. In: International Symposium on Leveraging Applications of Formal Methods, Verification and Validation, pp. 41–55 (2014)
8. Bianculli, D., Filieri, A., Ghezzi, C., Mandrioli, D., Rizzi, A.M.: Syntax-driven program verification of matching logic properties. In: Proceedings of FME Workshop on Formal Methods in Software Engineering (2015)
9. Blair, G., Bencomo, N., France, R.B.: Models @ run time. IEEE Comput. **42**(10), 22–27 (2009)
10. Calinescu, R., Ghezzi, C., Kwiatkowska, M., Mirandola, R.: Self-adaptive software needs quantitative verification at runtime. Commun. ACM **55**(9), 69–77 (2012)
11. Cheng, B., et al.: Software engineering for self-adaptive systems: a research roadmap. In: Software Engineering for Self-adaptive Systems, pp. 1–26. Springer, Berlin (2009)
12. Combemale, B., France, R., Jézéquel, J.M., Rumpe, B., Steel, J., Vojtisek, D.: Engineering Modeling Languages: Turning Domain Knowledge into Tools. CRC Press, Boca Raton (2016)
13. De Lemos, R., et al.: Software engineering for self-adaptive systems: a second research roadmap. In: Software Engineering for Self-adaptive Systems II, pp. 1–32. Springer, Berlin (2013)
14. Epifani, I., Ghezzi, C., Mirandola, R., Tamburrelli, G.: Model evolution by run-time parameter adaptation. In: IEEE 31st International Conference on Software Engineering, pp. 111–121 (2009)
15. Filieri, A., Ghezzi, C., Tamburrelli, G.: Run-time efficient probabilistic model checking. In: Proceedings of the 33rd International Conference on Software Engineering, pp. 341–350 (2011)
16. Filieri, A., Ghezzi, C., Tamburrelli, G.: A formal approach to adaptive software: continuous assurance of non-functional requirements. Form. Asp. Comput. **24**(2), 163–186 (2012)
17. Filieri, A., Tamburrelli, G., Ghezzi, C.: Supporting self-adaptation via quantitative verification and sensitivity analysis at run time. IEEE Trans. Softw. Eng. **42**(1), 75–99 (2016)
18. Ghezzi, C.: Dependability of adaptable and evolvable distributed systems. In: Bernardo, M., De Nicola, R., Hillston, J. (eds.) Formal Methods for the Quantitative Evaluation of Collective Adaptive Systems: 16th International School on Formal Methods for the Design of Computer, Communication, and Software Systems, SFM 2016, Bertinoro, June 20–24, 2016, pp. 36–60. Springer, Berlin (2016)
19. Ghezzi, C.: Of software and change. J. Softw. Evol. Process **29**, 1–14 (2017)
20. Ghezzi, C., Tamburrelli, G.: Reasoning on non-functional requirements for integrated services. In: Proceedings of 17th IEEE International Requirements Engineering Conference, pp. 69–78 (2009)
21. Ghezzi, C., Pinto Sales, L., Spoletini, P., Tamburrelli, G.: Managing non-functional uncertainty via model-driven adaptivity. In: Proceedings of the 2013 International Conference on Software Engineering, pp. 33–42 (2013)

22. Ghezzi, C., Sharifloo Molzam, A., Menghi, C.: Towards agile verification. In: Münch, J., Schmid, K. (eds.) Perspectives on the Future of Software Engineering: Essays in Honor of Dieter Rombach, pp. 31–47. Springer, Berlin (2013)
23. Ghezzi, C., Panzica La Manna, V., Motta, A., Tamburrelli, G.: Performance-driven dynamic service selection. Concurr. Comput. Pract. Exp. **27**(3), 633–650 (2015)
24. Hallsteinsen, S., Hinchey, M., Sooyong P., Schmid, K.: Dynamic software product lines. IEEE Comput. **41**(4), 93–95 (2008)
25. Kephart, J.O., Chess, D.M.: The vision of autonomic computing. IEEE Comput. **36**(1), 41–50 (2003)
26. Kramer, J., Magee, J.: The evolving philosophers problem: dynamic change management. IEEE Trans. Softw. Eng. **16**(11), 1293–1306 (1990)
27. Lehman, M.M., Belady, L.A.: Program Evolution: Processes of Software Change. Academic Press Professional, Inc., San Diego (1985)
28. Menghi, C.: Verifying incomplete and evolving specifications. In: Companion Proceedings of the 36th International Conference on Software Engineering, pp. 670–673 (2014)
29. Menghi, C., Spoletini, P., Ghezzi, C.: Dealing with incompleteness in automata-based model checking. In: FM 2016: Formal Methods - 21st International Symposium, Limassol, November 9–11, 2016, Proceedings, pp. 531–550 (2016)
30. Meyer, B.: Agile!: The Good, the Hype and the Ugly. Springer Science, Berlin (2014)
31. Parnas, D.L.: On the criteria to be used in decomposing systems into modules. Commun. ACM **15**(12), 1053–1058 (1972)
32. Parnas, D.L.: On the design and development of program families. IEEE Trans. Softw. Eng. **2**(1), 1–9 (1976)
33. Parnas, D.L.: A rational design process: How and why to fake it. IEEE Trans. Softw. Eng. **12**(2), 251–257 (1986)
34. Parnas, D.L.: Software aging. In: Proceedings of the 16th International Conference on Software Engineering, pp. 279–287 (1994)
35. Uchitel, S., Alrajeh, D., Ben-David, S., Braberman, V., Chechik, M., De Caso, G., D'Ippolito, N., Fischbein, D., Garbervetsky, D., Kramer, J., Russo, A., Sibay, G.: Supporting incremental behaviour model elaboration. Comput. Sci. Res. Dev. **28**(4), 279–293 (2013)
36. Vandewoude, Y., Ebraert, P., Berbers, Y., D'Hondt, T.: Tranquility: a low disruptive alternative to quiescence for ensuring safe dynamic updates. IEEE Trans. Softw. Eng. **33**(12), 856–868 (2007)
37. Zave, P., Jackson, M.: Four dark corners of requirements engineering. ACM Trans. Softw. Eng. Methodol. **6**(1), 1–30 (1997)

Open Access This chapter is licensed under the terms of the Creative Commons Attribution 4.0 International License (http://creativecommons.org/licenses/by/4.0/), which permits use, sharing, adaptation, distribution and reproduction in any medium or format, as long as you give appropriate credit to the original author(s) and the source, provide a link to the Creative Commons license and indicate if changes were made.

The images or other third party material in this chapter are included in the chapter's Creative Commons license, unless indicated otherwise in a credit line to the material. If material is not included in the chapter's Creative Commons license and your intended use is not permitted by statutory regulation or exceeds the permitted use, you will need to obtain permission directly from the copyright holder.