On Unifying Development Models and Runtime Models (Position Paper)

Thomas Vogel and Holger Giese
Hasso Plattner Institute, University of Potsdam, Germany
thomas.vogel@hpi.de holger.giese@hpi.de

Abstract. Models@run.time research primarily focuses on developing and using self-representations, that is, runtime models reflecting running software systems. Such models are the basis for feedback loops to monitor, analyze, and adapt these systems while the goal is typically to completely automate these feedback loops (cf. self-adaptation). This focus ignores (1) the beneficial use of runtime models for (manual) maintenance, which can already be observed in practice, and (2) the inevitable coexistence of self-adaptation and maintenance. Both issues require the integration of development (or maintenance) models and runtime models. In this position paper, we envision the unification of development and runtime models to systematically realize the integration and we discuss the benefits of this unification for addressing these issues with an illustrative example. We claim that Models@run.time research should broaden its focus to the unification to support an incremental adoption of runtime models from manual maintenance to automated self-adaptation, and the coexistence. Finally, we discuss Models@run.time challenges for achieving this claim.

1 Introduction

Models@run.time [4] research focuses on developing self-representations, that is, runtime models that are causally connected to running software systems, and feedback loops operating on these models to realize self-adaptation [3, 15]. Self-adaptation promises that software adjusts itself by automating and shifting some development, maintenance, and evolution activities to the runtime [12] but we cannot expect that this holds for *all* activities. This requires manual maintenance (and evolution) [9] of self-adaptive software, which has to go hand in hand with automated self-adaptation [1, 6]. However, the current focus of Models@run.time research ignores (1) the beneficial use of runtime models for (manual) maintenance, which can already be observed in practice, and (2) the inevitable coexistence of self-adaptation and maintenance. Both issues require the integration of development (or maintenance) models and runtime models.

In this position paper, we envision the unification of development and runtime models to systematically realize the integration. We use an illustrative example based on [5], in which a faulty application is adjusted to handle failures, to discuss both issues and how the unification may support them. In particular, the required adjustments can be achieved by maintenance, self-adaptation, or a coexistence of both. Similar to self-adaptation, maintenance can be considered as a feedback loop [11] such that we discuss the beneficial use of unified development and runtime models for maintenance, self-adaptation, and the coexistence using the feedback loop concepts of monitor, analyze, plan, and execute [8]. We claim that Models@run.time research should broaden its focus from pure self-representations of and embedded in running systems to this unification as it supports an incremental adoption of runtime models from maintenance to self-adaptation and the inevitable coexistence of self-adaptation and maintenance.

The rest of the paper is structured as follows. In Sec. 2, we outline the maintenance and self-adaptation feedback loops, their coexistence, and the state of the art in unifying development and runtime models for these loops. Using the illustrative example, we discuss traditional maintenance (Sec. 3) that incrementally adopts runtime models by unifying development and runtime models (Sec. 4 and 5) until self-adaptation (Sec. 6) and the coexistence (Sec. 7) are achieved. Finally, we conclude and discuss Models@run.time challenges for the envisioned unification.

2 Feedback Loops for Maintenance and Self-Adaptation

Fig. 1 depicts an overview of the individual feedback loops. Models@run.time research focuses on feedback loops for self-adaptation **(SA)**, that is, the runtime system monitors and analyzes itself, and if required, plans and executes its adaptation. Models@run.time research investigates the development and use of runtime models for these loops in runtime environments and already has achieved significant results [3, 15]. However, it has not addressed feedback loops for maintenance **(M)** regardless whether the system is self-adaptive or not. That is, engineers analyze and plan the software evolution in the development environment by using development models and they may interact with the runtime environment to monitor the running system and to execute updates. Though human-driven maintenance is often decoupled from the running system [9], it is conceivable that runtime and development models are exchanged between engineers and the system to seamlessly close the maintenance feedback loop. In general, self-adaptation and maintenance feedback loops may involve end users, for instance, to report bugs or execute a patch for maintenance, or to provide preferences for self-adaptation. Thus, involved humans **(H)** may interact with the runtime system or engineers ideally by exchanging user-friendly models.

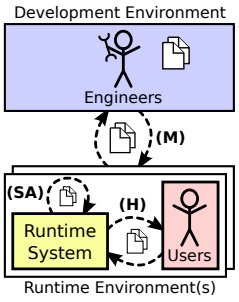

Fig. 1. Feedback Loops.

Besides the individual self-adaptation and maintenance feedback loops potentially involving users, both feedback loops are present and must coordinate if the system is self-adaptive [1, 6]. Hence, some of the maintenance activities have been automated and shifted from the development to the runtime environment, which leads to a blurring boundary between these environments [2]. For instance, coordination is needed to synchronize changes caused by self-adaptation with changes caused by maintenance. The coexistence requires the seamless exchange of runtime and development models between self-adaptation and maintenance feedback loops.

All this calls for a unification of development and runtime models to support seamless maintenance spanning the development and runtime environments and to enable the coexistence of self-adaptation and maintenance. The need for the coexistence has been recognized in [1, 2, 6] but without elaborating on runtime models. The state of the art in Models@run.time either provides approaches that exchange events between the development and runtime environments [13] or that allow engineers to upload new runtime models to the runtime environment [7, 16]. All of these approaches just use runtime models—as employed in the runtime environment—in the development environment and they do not relate or even unify them with development models.

3 Illustrative Example and Traditional Maintenance

The illustrative example is derived from [5]. It considers an application that uses a faulty library in a certain way. If this way of use leads to a failure, *workarounds* as alternative uses of the library are available that result in the same functionality but that avoid the failure. The rationale of workarounds is the intrinsic redundancy of libraries in providing variants of the same functionality. Exploiting this redundancy, the application must be patched to employ a workaround.

Scenario. Such a patch can be achieved by traditional maintenance, that is, human-driven maintenance processes decoupled from the running system [9, 14]. Considering maintenance as a feedback loop [11], we employ a subset of the EUREMA modeling language—originally designed to model feedback loops for self-adaptation [16]—to describe the maintenance scenario for the example (cf. Fig. 2). Hexagon block arrows describe activities, and solid arrows the control flow between activities. Rectangles represent artifacts (e.g., development and runtime models), and dotted arrows define the use of artifacts by activities (e.g., to **c**reate, **a**nnotate, **r**ead, and **w**rite a model). The colors in the figure highlight the roles performing the activities while some artifacts lie on the boundary between roles as these artifacts are shared among roles.

In Fig. 2, a user of the application detects a failure and creates a failure report in an issue/bug tracking system. This report only contains information that the user *can* provide based on her observations such as how she used the application and how the failure has reified. Engineers analyze such reports using a design model (incl. code) to identify the fault. Typically, engineers have implicit knowledge (i.e., a mental model) about the application such as the faulty library with its workarounds. This knowledge is the basis for developing a patch such that the application employs another workaround in using the library. Engineers release the patch by publishing it for download. Finally, users obtain and apply the patch to their installed applications.

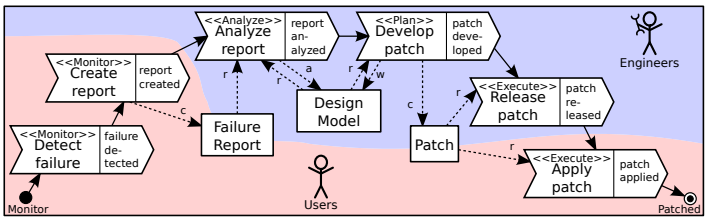

Fig. 2. Traditional Maintenance.

Observations. For the scenario of traditional maintenance, we observe that the feedback loop is divided in a monitor and execute part performed by users and in an analyze and plan part performed by engineers. Fixing the fault is based on design models, implicit knowledge of engineers, and the information provided by users. Particularly, the latter only refers to failures but not to faults as the user can only observe failures and is not aware of the faults or generally the internals of the application. Hence, the runtime information provided to the engineers is limited and not necessarily well-structured, which likely requires more efforts by engineers to identify the fault. However, artifacts such as user-written reports or patches are exchanged between engineers in the development environment and users in runtime environments.

4 Runtime Models for Monitoring in Maintenance

Based on the observations from traditional maintenance (Sec. 3), monitoring can be improved to provide richer and well-structured runtime information to engineers. One approach is to instrument the application with sensors, automate monitoring, and employ runtime models (cf. Fig 3).

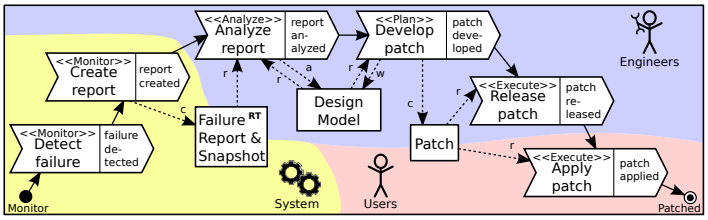

Fig. 3. Runtime Models for Monitoring in Maintenance.

Scenario. In contrast to users detecting and reporting failures, the system itself takes over these tasks. Besides freeing users from these tasks, the benefit of automated monitoring is that the system may report information internal to the system and not known to the users. For instance, architectural or configuration snapshots of the failing application can be created automatically and sent to engineers. Such snapshots provide richer and precise, and likely more useful information for engineers to identify the fault more quickly as user-created failure reports can only refer to the observable failure but not to the internal (indications of the) fault. If we consider such a Failure Report & Snapshot as a runtime (**RT**) model (cf. Fig. 3), the monitored information is well-structured and well-defined (e.g., by standardized exchange formats such as XMI and by metamodels) and engineers may directly employ model-based analysis techniques. Moreover, further specialized and abstract views of the running system can be efficiently created from a runtime model [17].

Observations. For this scenario, we can observe in practice that tools are already embedded into applications to automatically create and send crash reports to engineers (e.g., *Apport*, `https://wiki.ubuntu.com/Apport`). However, such tools work at the code level and require transformations of the reports to raise the abstraction level and to enable tool-supported analysis. In

this context, Models@run.time can be beneficial to provide abstract model-based views/reports that are more accessible for engineers and that can be directly analyzed with model-based analysis tools. In general, we think that Models@run.time can support the seamless transition between runtime system monitoring and engineers. Moreover, adopting Models@run.time for monitoring provides well-structured and well-defined information to ease and speed up the subsequent work by engineers. This requires that runtime and development models are designed in a unified manner such that both can be reasonably used at the same time by engineers.

5 Runtime Models for Execution in Maintenance

Likewise to disburdening users from the monitoring tasks (Sec. 4), the execute activity can be automated (i.e., instrumenting the application with effectors) and based on runtime models.

Scenario. As discussed for the previous scenario, a failure is reported by the system and engineers develop a patch (cf. Fig. 4). But this time, the user is not involved in executing the patch. In contrast, engineers release a patch as a runtime (**RT**) model describing the patch itself and how it should be executed. The system receives and automatically applies the patch. Being a runtime model, the patch can be analyzed or tested by the system at the model level, for instance, on the latest snapshot created by the automated monitoring activity (cf. Sec. 4), before it is actually applied. Thereby, the patch execution can be related to the monitored observations of the specific application instance since engineers typically develop patches for *all* instances of the application.

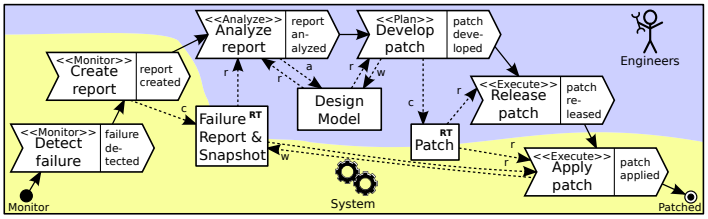

Fig. 4. Runtime Models for Execution in Maintenance.

Observations. In general, automating the execution disburdens the user from maintenance tasks and reduces the turnaround time of the maintenance process. The latter enables faster patches for all application instances. In practice, we can already observe that patches or generally updates are largely executed automatically, that is, users are notified about updates and only must agree to install them (e.g., in *Ubuntu*, `https://wiki.ubuntu.com/SoftwareUpdates`) or that updates are checked with local configurations in separate test environments before they are actually applied (e.g., [10]). Models@run.time can be beneficial to support the seamless transition from building patches in the development environment to the applicability analyses and execution in individual runtime environments. In particular, it can be beneficial to address the variability of (self-adaptive) application instances in heterogeneous runtime environments.

6 Runtime Models for Self-Adaptation

For our example, we may close the maintenance feedback loop in the system itself by further automating the analyze and plan activities. This results in a feedback loop for self-adaptation (cf. Fig. 5).

Scenario. As before, the system monitors itself to detect and report failures. The report is automatically analyzed to find applicable workarounds that are specified in the Workarounds runtime model, one of which will then be selected. From the selected workaround, an executable patch is created and finally applied by

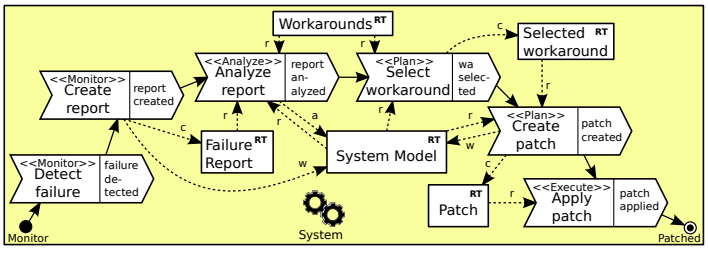

Fig. 5. Runtime Models for Self-Adaptation.

the system before the failure reaches the user, that is, the feedback loop masks the failure. These activities operate on the System Model—a self-representation of the running system—that is kept up-to-date by monitoring and that is the basis for automated analysis and planning.

Observations. Completely automating the feedback loop enables the immediate and in-situ handling of failures before users notice them, hence making the system resilient to failures. In the maintenance scenarios (Sections 3-5), the user is interrupted as the failure might impede the use of the system until engineers provide a patch. Generally, automating the analyze and plan activities requires two things. First, the system must have a self-representation of itself (cf. System Model) as a basis for the analysis and planning, which is achieved in maintenance by a Design Model plus optionally a system Snapshot. Second, the implicit knowledge of the engineers about the workarounds must be explicitly specified, for instance, in a runtime model as adaptation rules (cf. Workarounds). Besides its benefits for self-adaptation, Models@run.time can be beneficial in supporting an incremental transition of shifting some maintenance tasks to the self-adaptive system. This requires the seamless connection of development and runtime models, for instance, to exploit development-time knowledge such as design rationale for self-adaptation. We envision the unification of development and runtime models to achieve such a seamless connection.

7 Runtime Models for the Coexistence

Though self-adaptation with runtime models provides several benefits with respect to maintenance (cf. previous sections), maintenance does not become dispensable.

Scenario. In our example, the deployed Workarounds are specified by engineers (cf. Fig. 5) but these workarounds are likely not complete in the sense that they address *all* classes of failures (such as unforeseen ones). Hence, as shown in Fig. 6, a suitable workaround might not be found automatically (cf. bottom layer), which calls for maintenance activities (cf. top layer). In this case, an Extended failure report & System Model is created for engineers to specify new workarounds and update the workarounds deployed in the runtime system.

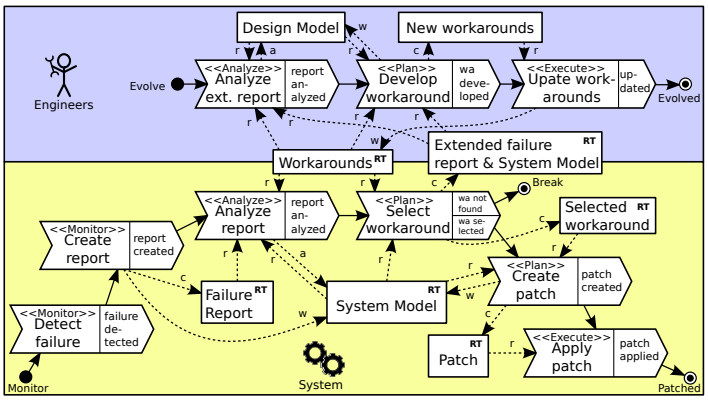

Fig. 6. Runtime Models for the Coexistence.

Observations. Combining self-adaptation and maintenance requires their coexistence such as when and how engineers evolve a self-adaptive system. For instance, to address a failure for which the deployed workarounds are not sufficient, engineers may directly develop and enact a patch as described in Fig. 4. Such an approach, however, competes with self-adaptation because the system may adapt itself while the patch is developed making the patch useless. Hence, the maintenance and self-adaptation feedback loops must be coordinated, for instance, by enacting maintenance changes via the self-adaptation feedback loop as shown in Fig. 6, that is, engineers provide new workarounds for the self-adaptation loop rather than directly patching the underlying system to handle the failure. Models@run.time are beneficial for the coexistence as they provide the flexibility in self-adaptation feedback loops for maintenance. Moreover, they provide the same benefits for exchanging models between engineers and the self-adaptive system as for automated monitoring and execution discussed in Sec. 4 and 5. But they also have to be unified with development models to simultaneously support self-adaptation and maintenance.

8 Conclusion and Challenges

In this paper, we have discussed the benefits and use of runtime models for traditional maintenance, maintenance that incrementally adopts Models@run.time principles, self-adaptation, and the coexistence of maintenance and self-adaptation. Leveraging these benefits requires a unification of development and runtime models. In our opinion, this unification should be more in the focus of Models@run.time research that is currently more concerned with self-representations and self-adaptation. We think that this broadened focus supports maintenance as well as an incremental adoption of Models@run.time principles to achieve the inevitable coexistence. Finally, we outline major challenges with respect to the unification of development and runtime models:

- How can we systematically derive runtime models from development models?
- How to co-design, co-evolve, and co-assure the self-adaptive system (product) and the processes for development, maintenance, and self-adaptation with Models@run.time?
- Can we find an MDE style that works for the development and runtime environment such as a common API with different capabilities? Which capabilities are key for each environment?
- How do interfaces of runtime models should be designed such that engineers can use and interact with them when maintaining a system employing Models@run.time?
- What are the specifics of runtime models for the individual MAPE steps of feedback loops supporting automated self-adaptation and manual maintenance?
- How can we efficiently exchange or synchronize models between development and runtime environments? Which formats and protocols are suitable for exchanging runtime models?
- How can Models@run.time support the maintenance of *multiple* runtime environments by *one* development environment, particularly, when each instance of the runtime system has a different configuration due to self-adaptation? How can we efficiently aggregate runtime models from all these instances, and distribute runtime models to them? How to extend this setting to software ecosystems with multiple development/runtime environments? In this context, what can we learn for Models@run.time from software product line research?
- Exchanging models between runtime and development environments, what are fundamental criteria (if any) distinguishing runtime models from development models?

References

1. Andersson, J., Baresi, L., Bencomo, N., de Lemos, R., Gorla, A., Inverardi, P., Vogel, T.: Software Engineering Processes for Self-Adaptive Systems. In: SEfSAS II, LNCS 7475, pp. 51–75. Springer (2013)
2. Baresi, L., Ghezzi, C.: The disappearing boundary between development-time and run-time. In: FoSER'10. pp. 17–22. ACM (2010)
3. Bencomo, N., France, R., Cheng, B., Assmann, U. (eds.): Models@run.time, LNCS 8378. Springer (2014)
4. Blair, G., Bencomo, N., France, R.B.: Models@ run.time. Computer 42(10), 22–27 (2009)
5. Carzaniga, A., Gorla, A., Mattavelli, A., Perino, N., Pezzè, M.: Automatic Recovery from Runtime Failures. In: ICSE'13. pp. 782–791. IEEE (2013)
6. Gacek, C., Giese, H., Hadar, E.: Friends or Foes? – A Conceptual Analysis of Self-Adaptation and IT Change Management. In: SEAMS'08. pp. 121–128. ACM (2008)
7. Iftikhar, M.U., Weyns, D.: Activforms: Active formal models for self-adaptation. In: SEAMS'14. pp. 125–134. ACM (2014)
8. Kephart, J., Chess, D.: The vision of autonomic computing. Computer 36(1) (2003)
9. Kitchenham, B., et. al.: Towards an ontology of software maintenance. Journal of Software Maintenance: Research and Practice 11(6), 365–389 (1999)
10. Kösegi, A., Nerding, R.: SAP Change and Transport Management. SAP Press (2009)
11. Lehman, M.M.: Feedback in the software evolution process. Information and Software Technology, Special Issue on Software Maintenance 38(11), 681–686 (1996)
12. de Lemos, R., Giese, H., Müller, H.A., Shaw, M., et al.: Software Engineering for Self-Adaptive Systems: A second Research Roadmap. In: SEfSAS II, LNCS 7475, pp. 1–32. Springer (2013)
13. Morin, B., Ledoux, T., Hassine, M.B., Chauvel, F., Barais, O., Jézéquel, J.M.: Unifying Runtime Adaptation and Design Evolution. In: CIT'2009 – Vol. 02. pp. 104–109. IEEE (2009)
14. Sommerville, I.: Software Engineering. Addison-Wesley, 9 edn. (2010)
15. Szvetits, M., Zdun, U.: Systematic literature review of the objectives, techniques, kinds, and architectures of models at runtime. Software & Systems Modeling pp. 1–39 (2013)
16. Vogel, T., Giese, H.: Model-driven engineering of self-adaptive software with eurema. ACM Trans. Auton. Adapt. Syst. 8(4), 18:1–18:33 (2014)
17. Vogel, T., Neumann, S., Hildebrandt, S., Giese, H., Becker, B.: Incremental model synchronization for efficient run-time monitoring. In: Models in Software Engineering, LNCS 6002, pp. 124–139. Springer (2010)