Classifying Distributed Self-* Systems Based on Runtime Models and Their Coupling

Sebastian Wätzoldt and Holger Giese

[sebastian.waetzoldt|holger.giese]@hpi.de
Hasso Plattner Institute for Software Systems Engineering
University of Potsdam, Germany

Abstract. Different kinds of self-* systems ranging from autonomous self-organizing to hierarchical self-adaptive systems have been developed in the past. However, today there are no clear technical criteria how to classify distributed self-* systems within the resulting design spectrum. In this paper, we provide such a classification by looking on runtime models and their coupling. As runtime models capture the shared knowledge employed by feedback loops, at first, we provide an improved runtime model categorization. With such a basis, we subsequently derive impact relations describing the coupling of runtime models. Finally, we show that the existence of complex impact relations can be employed to describe the spectrum of self-* systems.

1 Introduction

There are different kinds of self-* systems ranging from autonomous self-organizing to hierarchical self-adaptive systems (cf. [4]). Besides single feedback loops controlling the adaptation of the core system, also multiple feedback loops realizing different adaptation concerns and distributed, collaborating feedback loops are employed to achieve the desired self-* behavior for the whole system (cf. [2]). However, today, there are no clear technical criteria how to classify distributed self-* systems within the resulting design spectrum [9, 19, 21].

Runtime models are an abstract representation of the running software system that represent the knowledge employed in the MAPE-K feedback loop approach [10], which is characteristic for self-* behavior (cf. [2]). Runtime Models are causally connected to the running system [1]. Therefore, it seems to be a good idea to consider runtime models as starting point, when looking for a classification of the design spectrum for distributed self-* systems.

In this paper, we provide a classification for distributed self-* system by looking on runtime models and their coupling. As the runtime models capture the shared knowledge and the coupling reflects the employed feedback loops, we can derive the classification as follows: After discussing the state of the art in the next Section 2, we provide an improved runtime model categorization in Section 3. Subsequently, we derive several impact relations describing the coupling of runtime models in Section 4 on basis of the introduced runtime model categorization. Furthermore, we employ complex impact relations to characterize the spectrum of distributed self-* systems in Section 5, where we outline typical cases from literature. Finally, we conclude in Section 6.

2 State of the Art

In this section, we discuss related approaches that explicitly consider feedback loop (activities) and runtime models as first class entities in the design and execution for self-* systems.

There are several frameworks that support the handling of feedback loop activities and the corresponding knowledge base. For the automotive domain, Zeller et al. [21] presented a control architecture that handles hierarchically arranged feedback loops, where each loop maintains an individual piece of knowledge according to its adaptation purpose. Additionally, feedback loops on a higher hierarchical layer have a unified, aggregated view on the whole knowledge of the layer below. In this approach, the feedback loop and knowledge dependency are rather fix and implicitly encoded in the formal model over the hierarchy.

In [9], the authors presented a formal approach called ActivFORMS, where both, feedback loop activities and knowledge, can be represented with timed automata. Therefore, dependencies between activities and knowledge are encoded in the signal handling of the time automata formalism and thus, are implicitly available. Furthermore, the authors presented a runtime goal verification mechanism, which implies an explicit runtime model handling that is done by special management components. Also other frameworks, as for example Rainbow [6], introduce model handling mechanism providing access to runtime models and trace dependencies to corresponding adaptation activities. However, the mentioned approaches do not explicitly model nor capture runtime model and adaptation activity dependencies as first class entities. As a consequence, the resulting self-* systems have limitations concerning impact analysis. Such an analysis may enable the tracing of goals and corresponding activities (timed automata) in [9] or may visualize the control dependencies between feedback loops in hierarchical systems as in [21].

The explicitly consideration of dependencies between feedback loops in self-adaptive system has been discussed in [2, 4]. Based on this research, a pattern catalog for decentralized feedback loops, which include a discussion about dependencies between adaptation activities is presented in [19]. However, dependencies between the knowledge (runtime models), possible access patterns to the knowledge and the arising runtime model impact relations have not been considered.

We are aware of these limitations and presented the EUREMA modeling language (cf. [17]), where we explicitly model dependencies between adaptation activities and the access to runtime model using model management techniques from the MDE. Furthermore, we described a first categorization of runtime model in [17, 18] and derive requirements for runtime models in [16]. However, in this paper we significantly extend our preliminary runtime model categorization, describe runtime model characteristics and define clear criteria to distinguish between different runtime model types. On basis of the extended runtime model categorization, to the best of our knowledge, no existing approach successive derive impact relations based on the effect propagation that arise by accessing runtime models. Furthermore, we use the explicit modeled impact relations to systematically classify the design spectrum of self-* systems.

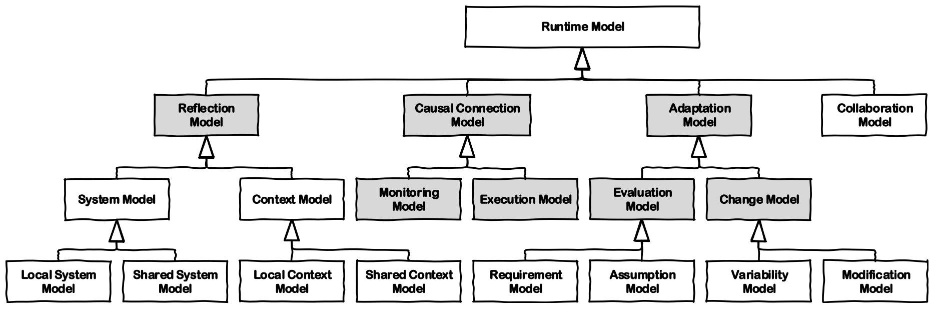


Fig. 1. Runtime model hierarchy (gray model types are from former work [17]).

3 Categorization of Runtime Models

In this section, we provide definitions and a categorization for runtime models that is used as foundation to derive different impact relations in the next Section 4. We consider four main categories of runtime models as depicted in Figure 1. Reflection Models describe aspects of interest from the running system and system context on a higher level of abstraction. Adaptation Models contain requirements as well as the configuration space of the adaptive system and are mainly used by the analyze and plan activity as depicted in Figure 2. Causal Connection Models are responsible to establish the causal connection to the running system and therefore primary used during the monitoring and executing feedback loop activities as shown on the bottom of Figure 2. Finally, Collaboration Models are important for the coordinated interaction of distributed feedback loops. We use the runtime models definition from our former work in [7, p. 13] and define each runtime model category of Figure 1 as follows:

Runtime Reflection Model: Runtime Reflection Models describe concerns that are related to the running system in System Models as well as system context in Context Models. System Models are directly causal connected and provide reflected *architectural* and *behavioral* views of the system for the key points of interest. The structural system parts are often modeled in form of component diagrams as in [1, 6, 15], whereas behavioral system aspects are often modeled as processes (e.g., business process diagrams) or automata [9]. As fine-grain classification of System Models, we distinguish between Local System Models and Shared System Models. While runtime model manipulations in Local System Models cause local system changes (direct causal connection), changes in the Shared System Models affect runtime models in collaborating feedback loops (indirect causal connection) as discussed for the IR–5 impact relation in Section 4.

Runtime Context Models describe the context of a system that is defined by [5] as *"any information that can be used to characterize the situation of an entity"*. According to this definition, we consider the adaptable system as the *entity*. Additionally, we distinguish between system context and system environment. The system context can be captured by the system, is a subset of the system environment and maintained in runtime Local Context Models, whereas the system environment is a superset of the system context that additionally contains all not detectable parts of the system's surroundings. As a consequence of the context definition, Context Models and System Models are disjoint. Information in the Local Context Models can be retrieved by the system itself (e.g., a monitor activity senses and updates the model), whereas Shared Context Models capture additional information about other systems that is retrieved during collaboration.

Runtime Adaptation Model: Adaptation Models describe the possible solution space of the system by (1) declarative requirements and (2) potential variants of the adaptable software system. (1) The overall system specification are determined in Evaluation Models, which describe functional and non-functional properties. As defined in [20, p. 27], the system specification consists of requirements and assumptions about the context. Therefore, we capture the system specification in Requirement Models and Assumption Models accordingly. (2) Beside the defined solution space of the system in the Evaluation Models, Change Models describe possible solutions, where the system might adapt to. Variability Models explicitly model the possible solution space (a subset of all valid solutions) similar to software product lines. During the adaptation process, one configuration can be chosen that will cause predefined effects on the system (e.g., exchange of a component). In contrast, Modification Models implicitly model the possible solution space (and thus a superset of all valid solutions) by defining all permitted modifications of the Reflection Models. Due to the causal connection, applying Change Models to System Models will enforce the desired change in the system.

Runtime Causal Connection Model: Causal Connection Models establish the causal connection to the running system. Monitoring Models retrieve information from the running system and update the information in the corresponding Reflection Model, whereas Execution Models propagate changes from the Reflection Model to the running system. Therefore, Monitoring Models describe the mapping of system-level observations to the abstraction level of the Reflection Models and Execution Models translate runtime model adaptations to system adaptations. Usually, causal connection models depend on implementation specifics in the adaptable software and can be realized by MDE techniques such as graph transformation (cf. [17]). Note: The causal connection is established between runtime models from the adaptation engine to the adaptable software as well as between different layers (if existing) inside the adaptation engine (cf. IR–4 in Section 4). As a consequence, the *sensors* and *effectors* in Figure 2 are realized by the monitoring and execution models.

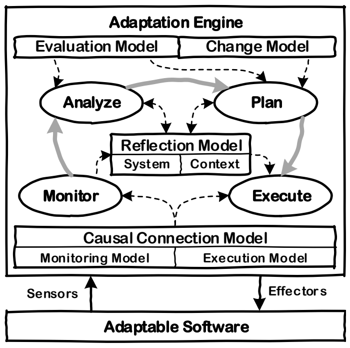


Fig. 2. MAPE feedback loop using different runtime model types.

Runtime Collaboration Model: Feedback loop interaction arises if multiple feedback loops are considered for different concerns such as reliability and performance (cf. [10, 15]) or for distributed systems, where individual parts have to interact with each other. Such aspects are modeled in Collaboration Models that typically comprise interaction protocols describing the choreography between feedback loops. Therefore, Collaboration Models establish and maintain a connection between different feedback loops.

In summary, the runtime model categorization supports a clear description of well-defined system concerns using distinct, loosely coupled runtime models that are accessed and arranged around feedback loop activities as depicted in Figure 2. The categorization enables the identification of dependencies between runtime models and feedback loop activities as described in the next Section.

4 Impact Relations of Runtime Models

In this section, we identify five impact relations between runtime models. Impact relations arise if different feedback loop activities access the same runtime model over time, which leads to an indirect coupling over the shared knowledge and potentially impacts independent running parts of the adaptation engine.

All impact relations are depicted in Figure 3, where runtime models are modeled as rectangles labeled with the corresponding runtime model type. Furthermore, feedback loop activities are modeled as ellipses and dependencies are arrows between activities and runtime models. We distinguish for dependency types, namely control flow between activities (gray arrows), impact relations (black arrows), collaboration between feedback loop activities (arrow starting with a ball), and model access operation (dashed arrows). In [17], we describe how feedback loop activities modify and access runtime models via predefined model operations that are **c**reating, **d**estroying, **w**riting, **r**eading and **a**nnotating. While reading a runtime model has no side effect, other model operations will cause a modification of the runtime model.[1] We extend the set of runtime model operations by a **ref**lect and **af**fect model operation. Normally, the monitor activity uses the reflect model operation for sensing information of interest from the lower layer and storing it as abstract representation in the system runtime model of the upper layer. As counterpart, the execute activity propagate changes in the system runtime model via the affect model operation to the layer below. Therefore, the reflect/affect operations work in the same way as the sensors/effectors in Figure 2 and are realized by the runtime Causal Connection Models (cf. Section 3). While IR–1 and IR–2 describe impact relations inside a feedback loop, the impact relations IR–3, IR–4, and IR–5 arise between multiple feedback loops.

IR–1 ***Intra-Runtime-Model:*** We can identify two versions (I, II) for a *Intra-Runtime-Model* impact relation (cf. upper left example in Figure 3). (I) A runtime model is accessed by an activity twice via a (1) read model operation first and (2) modify model operation afterwards. (II) A runtime model is accessed by two successive activities within one feedback loop, where the former activity (1) modifies and the following activity (2) reads the runtime model. The *Intra-Runtime-Model* impact relation typically arise for the (I) variant, if one activity updates the runtime model with information as for example during the monitoring or after an analysis step. The (II) variant arises if activities in the feedback loop successively use annotated information in the runtime model as it is normally the case for the planning activity after the analysis of the system model finished.

[1] For the rest of this paper, it is sufficient that we distinguish between read (r) and **m**odify (c,d,w,a) model operations. Modifying a runtime model includes the reading of it.

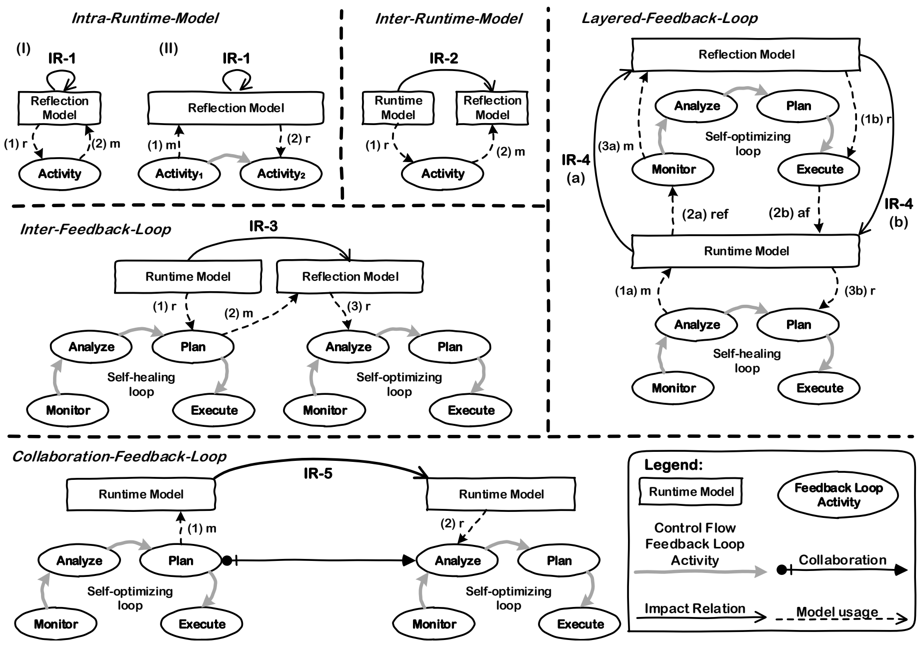


Fig. 3. Impact relations between runtime models.

IR–2 ***Inter-Runtime-Model:*** While IR–1 considers impact relations within the same runtime model, *Inter-Runtime-Model* impact relations determine the coupling between different runtime models (cf. upper middle example in Figure 3) and are characterized by: (1) At least one read access by an activity on an arbitrary runtime model type, followed by (2) a modify model operation on a reflection model by the same activity. Additionally, there must be at least two different runtime models (otherwise it is a IR–1) and the modified model is always a Reflection Model. As an example for a IR–2 impact relation, a monitor activity reads a causal connection model, senses information from the running software system and annotates the information in the corresponding system model.

IR–3 ***Inter-Feedback-Loop:*** Normally, multiple feedback loops are used for handling different adaptation concerns of the system. However, even if the feedback loops are conceptually disjunct, an indirect coupling is possible over the available runtime models. Therefore, the feedback loops may indirectly influence each other over the coupled knowledge base. The *Inter-Feedback-Loop* impact relation explicitly identifies the coupling over shared runtime models. We define an *Inter-Feedback-Loop* impact relation by three characteristic runtime model accesses (cf. middle left example in Figure 3). (1) At least one read access by an activity on an arbitrary runtime model to gather required knowledge. After the activity perform its task, it (2) modifies the Reflection Model that is (3) read afterwards by another activity. The both activities must be independent (e.g., there is no control flow in between), otherwise it is a combination of an IR–1 and IR–2.

IR–4 ***Layered-Feedback-Loop:*** While an IR–3 describes the impact relation between feedback loops on the same layer, the *Layered-Feedback-Loop* impact relation considers dependencies between hierarchies of feedback loops. With the help of the reflect and affect model operations, we can define a sensing (variant (a)) and effecting (variant (b)) *Layered-Feedback-Loop* impact relation as follows (cf. upper right example in Figure 3): (1a) There is at least one read access of an arbitrary activity from the lower layer. Additionally, a monitor activity in the upper layer must (2a) reflect the accessed runtime model of step (1a) from the lower layer. Finally, the reflected information become available for activities in the upper layer by (3a) modifying the corresponding reflection model. The effecting *Layered-Feedback-Loop* impact relation variant (b) is analog to the sensing variant but in the other direction (from the upper layer to the lower layer) and using the affect model operation in step (2b).

IR–5 ***Collaboration-Feedback-Loop:*** The *Collaboration-Feedback-Loop* impact relation identifies distributed runtime model dependencies in collaborating feedback loops. In a collaboration, feedback loops must coordinate each other (e.g., by a communication protocol) and maintain their own (local) runtime models. Although this impact relation implies the most indirect propagation of runtime model change effects due to the distribution aspect, we can determine it very precisely by means of the runtime model types access of the activities within the feedback loops. Therefore, this impact relation significantly distinguishes from IR–3. A *Collaboration-Feedback-Loop* impact relation arise for the following access characteristics (cf. example on the bottom in Figure 3): The feedback loop activity, which want to share knowledge (sender), (1) modifies an arbitrary runtime model. On basis of the updated knowledge, the sender performs its task including the defined coordination[2] and sends the knowledge to the other feedback loop activities (receivers) accordingly. The knowledge is saved into the runtime model of the receivers, which can be (2) read afterwards by the activity. Furthermore, all activities participating at the collaboration are not coupled via the control flow, otherwise it is an IR–1 impact relation. Additionally, we consider only direct (explicit) collaborations over the Collaboration Model. Due to the lack of a global, unified knowledge in distributed systems, influencing system parts over the context (e.g., by setting marks in the real environment) that is independently sensed by another system and therefore may influence the behavior, are not considered.

In summary, we can use the five presented impact relations for further effect propagation analysis. For doing so, we combine the impact relations IR–3, IR–4, and IR–5 with the impact relations IR–1 and IR–2 to transitively describe the influence of runtime model manipulations. For example, combining IR–3 and IR–2 may cause an effect propagation from the modified reflection model in the IR–3 relation to a modification of another system model via the IR–2 relation

[2] Activities participating in the collaboration must know the choreography of the joint interaction that is described in the Collaboration Model. However, the adaptation engine is responsible for the physical transmission of the data that is not further considered in this paper.

(cf. IR–3 + IR–2 in Figure 3). As a consequence, we are able to investigate the transitive closure for all possible effect propagations in the system that is used in the next section to classify distributed self-* systems.

5 Classification of Distributed Self-* Systems

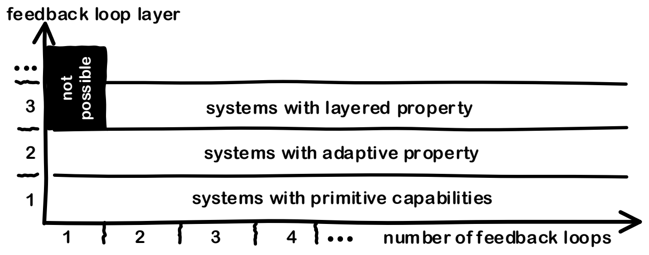
In this section, we describe the spectrum for self-* systems ranging from self-/context-aware systems over hierarchical self-adaptive systems to self-organizing distributed systems. On basis of the overview and categorization of several self-* properties provided by Salehie et al. [14, pp. 3-7], we discuss five variants of self-* system properties, namely, *primitive*, *adaptive*, *layered*, *hierarchical*, and *distributed*. A hierarchy of these properties is depicted in Figure 4.

Primitive Capabilities: Systems with one layer have, in the most cases, only the runtime model impact relations IR–1 and IR–2. Depending on the accessed runtime model category within the impact relation, such system can have different primitive capabilities [13, 14] as for example self-awareness (read model operation on the runtime System Model), context-awareness (read on the runtime Context Model) and requirement-awareness (read on the runtime Evaluation Model). Therefore, we can clearly identify the IR–1 or IR–2 impact relation that corresponds to the primitive *-awareness capability.

Adaptive Property: Systems with two layers are able to separate the domain logic and the adaptation logic as proposed in [10] and depicted in Figure 2. Beside the primitive capabilities of such systems, they have an adaptation engine on the higher layer, which usually consists of one single feedback loop that has IR–4 impact relations for sensing and effecting the adaptable software at the lower layer. Typically, such systems mostly realize one adaptation concern such as self-configuration, self-healing or self-optimizing and therefore, are characterized as adaptive systems. In the same way, we transitively derive impact relations for the description of propagation effects, we can combine the IR–4 characteristic of adaptive systems with other impact relations (e.g., IR–2). Therefore, the occurrence of different impact relations define clearly the system type as for example an IR–4 and an additional IR–2 relation type with a modify on a System Model consequently describes a self-adaptive system.

Layered Property: An increasing number of adaptation concerns, realized in separated feedback loops, might go hand in hand with a growing number of adaptation layers ensuring a proper system design as proposed in the reference architecture for self-adaptive systems [12]. This leads to an accumulated number of IR–4 impact relations between feedback loops for each layer, where each layer explicitly reflects/affects the layer below. The impact relation graph for the IR–4 impact relations (transitive closure) goes hand in hand with the layer structure. Other impact relation as for example IR–3 occur if additional feedback loops are used on the same layer. Examples for a layered systems are described in [3, 8].

Hierarchical Property: On basis of the impact relations IR–3 and IR–4, we conceptually distinguish between hierarchical controlled systems and layered adaptation. In general, hierarchies allow a decomposition of adaptation concerns in different abstraction levels. The difference between hierarchical control and layered adaptation is the impact relation type. Typically, hierarchical controller



feedback loop layer
...
3
2
1
not possible
systems with layered property
systems with adaptive property
systems with primitive capabilities
1
2
3
4
...
number of feedback loops

Fig. 4. Overview about adaptive and layered system properties.

are designed to interact with each other to well defined in-/output interfaces, whereas this idea can be transfered to feedback loop interaction as in [21]. However, hierarchical systems are characterized by a predominantly use of IR–3 impact relations that hierarchically couples the involved feedback loops. Consequently, hierarchical systems often interact via a predefined interface semantic instead of the reflect/affect model operations (cf. IR–3 and IR–4 in Section 4).

Distributed Property: Another direction of handling an increasing number of adaptation concerns are agent-based and self-organizing systems. Typically, each system part (e.g., an agent) is very simple structured with one or two layers and maintains its own local context. Such systems achieve higher self-* properties due to collaboration. In many cases, the collaborations are not fix and may change arbitrary often depending on the degree of autonomy. For example, agents may form distinct groups realizing a special adaptation concern. As a consequence, a high number of IR–5 impact relations as well as a low number of IR–3, IR–4 impact relations are a key indicator for self-organizing systems. An example for increasing the reliability of a self-organizing system using a variable number of software agents is described in [11].

In reality, we find several combinations of layered, hierarchical and distributed aspects between feedback loops that opens a large variety of systems. However, the impact relations describe the coupled knowledge between feedback loops and therefore the possible effect propagation in the system as well as allow a clear classification of self-* systems.

6 Conclusion

In this paper, we presented an approach to classify distributed self-* systems variants according to the occurrence of complex impact relations between runtime models. A combination of the discussed five basic impact relations allows the description of multilevel effect propagations over a coupled shared knowledge base. The impact relations are derived on basis of an improved runtime model categorization, where each runtime model type is clearly characterized. Furthermore, we can precisely assign primitive *-awareness properties to concrete accesses to a special runtime model type.

As next steps, we want to use the impact analysis to realize distributed self-* systems with the help of a model management approach that handles all runtime models, synchronizes the access to the models via multiple, distributed feedback loops activities and maintains different runtime model versions.

References

1. Blair, G., Bencomo, N., France, R.B.: Models@run.time. Computer 42(10), 22–27 (2009)
2. Brun, Y., Serugendo, G.D.M., et al.: Engineering Self-Adaptive Systems through Feedback Loops. In: SEfSAS. LNCS, vol. 5525, pp. 48–70. Springer (2009)
3. Burmester, S., Giese, H., Münch, E., Oberschelp, O., Klein, F., et al.: Tool Support for the Design of Self-Optimizing Mechatronic Multi-Agent Systems. International Journal on Software Tools for Technology Transfer 10(3), 207–222 (2008)
4. Cheng, B.H., de Lemos, R., Giese, H., et al.: Software Engineering for Self-Adaptive Systems: A Research Roadmap. In: SEfSAS. LNCS, vol. 5525, pp. 1–26 (2009)
5. Dey, A.K.: Understanding and Using Context. Personal Ubiquitous Comput. 5(1), 4–7 (2001)
6. Garlan, D., Cheng, S.W., Huang, A.C., et al.: Rainbow: Architecture-Based Self-Adaptation with Reusable Infrastructure. Computer 37(10), 46–54 (2004)
7. Giese, H., Bencomo, N., Pasquale, L., Ramirez, A., Inverardi, P., Wätzoldt, S., Clarke, S.: Living with uncertainty in the age of runtime models. In: Models@run.time, LNCS, vol. 8378, pp. 47–100. Springer (2014)
8. Giese, H., Schäfer, W.: Model-Driven Development of Safe Self-Optimizing Mechatronic Systems with MechatronicUML. In: Assurances for Self-Adaptive Systems, LNCS, vol. 7740, pp. 152–186. Springer (2013)
9. Iftikhar, M.U., Weyns, D.: Activforms: Active formal models for self-adaptation. In: SEAMS. pp. 125–134. ACM (2014)
10. Kephart, J.O., Chess, D.: The Vision of Autonomic Computing. Computer 36(1), 41–50 (2003)
11. Klein, F., Tichy, M.: Building Reliable Systems based on Self-Organizing Multi-Agent Systems. In: SELMAS. pp. 51–58. ACM Press (2006)
12. Kramer, J., Magee, J.: Self-Managed Systems: an Architectural Challenge. In: FOSE. pp. 259–268. IEEE Computer Society (2007)
13. Salehie, M., Tahvildari, L.: Autonomic computing: emerging trends and open problems. In: DEAS. pp. 1–7. ACM (2005)
14. Salehie, M., Tahvildari, L.: Self-adaptive software: Landscape and research challenges. ACM Trans. Auton. Adapt. Syst. 4(2), 1–42 (2009)
15. Vogel, T., Giese, H.: Adaptation and Abstract Runtime Models. In: SEAMS. pp. 39–48. ACM (2010)
16. Vogel, T., Giese, H.: Requirements and Assessment of Languages and Frameworks for Adaptation Models. In: Workshops and Symposia at MoDELS. LNCS, vol. 7167, pp. 167–182. Springer (2012)
17. 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)
18. Vogel, T., Seibel, A., Giese, H.: Toward Megamodels at Runtime. In: Models@run.time. CEUR Workshop Proceedings, vol. 641, pp. 13–24. CEUR-WS.org (2010)
19. Weyns, D., Schmerl, B., Grassi, V., Malek, S., Mirandola, R., Prehofer, C., Wuttke, J., Andersson, J., Giese, H., et al.: On Patterns for Decentralized Control in Self-Adaptive Systems. In: SEfSAS II, LNCS, vol. 7475, pp. 76–107. Springer (2013)
20. Zave, P., Jackson, M.: Four dark corners of requirements engineering. ACM Trans. Softw. Eng. Methodol. 6(1), 1–30 (1997)
21. Zeller, M., Prehofer, C.: A Multi-Layered Control Approach for Self-Adaptation in Automotive Embedded Systems. In: Advances in Software Engineering. vol. 2012, p. 15 (2012)