Towards software architecture runtime models for continuous adaptive monitoring

Thomas Brand
Hasso Plattner Institute, University of Potsdam
Potsdam, Germany
thomas.brand@hpi.uni-potsdam.de

Holger Giese
Hasso Plattner Institute, University of Potsdam
Potsdam, Germany
holger.giese@hpi.uni-potsdam.de

ABSTRACT

A software architecture runtime model provides an abstraction that allows to reason about a running system. For example, a self-adaptive system can employ the model to detect phenomena which make an adaptation beneficial. O ver t ime o ther p henomena can become interesting and thus make the monitoring of different system properties necessary. Typically properties are declared in a meta-model as part of application specific model element classifiers. In this case adding new properties requires the creation of a new runtime model instance based on the updated meta-model version. In contrast, a more flexible approach allows altering the set of properties in the runtime model without creating a new model instance and thus without interrupting the phenomena detection process. In this paper we elaborate requirements for a runtime model modeling language which shall enable continuous adaptive monitoring.

Reference Format:
Thomas Brand and Holger Giese. 2018. Towards software architecture runtime models for continuous adaptive monitoring. In *Proceedings of 13th International Workshop on Models@run.time (MRT 2018).* 6 pages.

1 INTRODUCTION

A runtime model which abstracts a running system to its architecture is a common practice for reasoning about the system [14]. The runtime model is maintained through monitoring to reflect relevant system changes in the model. A key question is, what information to obtain and represent in the model [5], especially as advances in information technology allow capturing more and more pieces of data. The answer to this question needs to consider the purpose for which the model is maintained as well as cost-effectiveness [12]. However, over time the information demand, which the runtime model needs to satisfy, can change regarding the level of abstraction, that is the set of monitorable properties, and for which of those up-to-date monitoring results are currently beneficial.

Triggers for such changes might for example be the temporary exploration of new features through machine learning, altered policies or monitoring results that lead to obtaining and incorporating additional information [6]. Adaptive monitoring allows focusing the attention according to the information demand by altering the set of monitorable properties and dynamically allocating monitoring effort to produce only currently relevant monitoring results.

Besides changing information demands also the represented running system can evolve over time, for example, due to the deployment of a new software component release. This can require the adaptive monitoring to maintain new kinds of information in the runtime model, too. Consequently, the runtime model also needs to evolve accordingly with the running system [3].

A software system can be built for continuous operation without downtime even when it evolves. Its runtime model needs to support this evolution without interruption to enable continuous adaptive monitoring and hence continuous phenomena detection and reasoning about the system. This is especially relevant if evolution occurs frequently for example caused by the currently fostered very short software release cycles and experimentation in software product development [7].

An idea to enable continuous adaptive monitoring is to utilize a runtime model modeling language which supports the co-evolution of the system and the model as well as changing information demands without the need to re-instantiate the runtime model and thereby avoiding an interruption. Such a modeling language with a reusable implementation to create and maintain runtime models would also reduce the effort to employ them and likely foster their use. Thus, we want to find out more about the related requirements.

Our contribution with this paper is a set of elaborated requirements for a modeling language to create and maintain evolvable runtime models that support continuous adaptive monitoring of a running system. For an prospective approach we describe different illustrative scenarios from which we derive and generalize the requirements. Additionally, we investigate two state-of-the-art approaches regarding their support for the scenarios and requirements to identify beneficial and missing features.

To the knowledge of the authors no reusable modeling language has been described and evaluated for runtime models that support continuous adaptive monitoring. Nor have requirements for such a language been elaborated before. Thus, this paper can be seen as a step towards software architecture runtime models for continuous adaptive monitoring.

In Section 2 the terminology used in this paper is introduced. In Section 3 we describe the scenarios and in Section 4 the derived requirements. Then we investigate two state-of-the-art approaches in Section 5 regarding their support for scenarios and requirements and outline a prospective runtime model modeling language to support continuous adaptive monitoring. Finally, we conclude and provide an outlook regarding future work in Section 6.

2 BACKGROUND

In this section we introduce the terminology used in this paper. A *runtime model* is defined as "an abstraction of a running system that is being manipulated at runtime for a specific purpose" [2]. There are different types of runtime models. This paper considers software architecture runtime models [14]. Those runtime models are graph-based data structures. They allow storing, querying and accessing configurational as well as operational information about the running system. Configurational information comprises the system parameters and the structure of the system with its parts and relationships. Operational information is a specific behavioral runtime aspect. It is caused or produced by the running system. An example is a component invocation count for a certain time window. Overall runtime models help "to handle the complexity of adapting or generally managing running software systems" [15].

A *modeling language* is constituted by "the structure, terms, notations, syntax, semantics, and integrity rules that are used to express a model" [10]. One way to describe a modeling language is by modeling it. The model used to model the modeling language is called meta-model. Typically, classifiers for model elements are defined in the language meta-model. A set of classifiers which belong together shall be called classifier system. An implementation of the modeling language allows to programmatically create and maintain language conform models. For example, the Eclipse Modeling Framework (EMF) allows generating a modeling language implementation from the language meta-model [13]. Typically language changes require repeating the code generation and re-instantiating the model by reloading or recreating it with the new implementation release.

As an ongoing abstraction of a running system a runtime model changes frequently in comparison to design time models. A change in a runtime model can, for example, be triggered when an automated control loop temporarily increases the value of a system configuration parameter to support the current workload of the system. Such mostly short-term and easily reversible adaptations shall be considered as ordinary runtime model changes. In contrast, evolutionary runtime model changes shall be those which involve defining a new or altering an existing classifier for runtime model elements. This usually requires a new release of the runtime modeling language and its implementation as well as a re-instantiation of the model. In this paper we will discuss another approach to change a runtime model classifier system and evolve the runtime model without this kind of interruption. Possible triggers for the runtime model evolution shall be changes in the information demand regarding the set of monitorable properties or the evolution of the represented system, for example, when a new kind of component is introduced in the system as a long-term enhancement. It shall be noted that in general the co-evolution of a runtime model and the represented running system is a relevant subject in the runtime model research field [3].

Adaptive monitoring helps to control the monitoring effort and focus on the information that is currently valuable. With regard to runtime models we consider two dimensions of monitoring adaptation. The first dimension is to change the level of abstraction for the runtime model or individual parts of it by altering the set of monitorable system properties. This comprises properties for configurational and operational information. The second dimension is to selectively enable and disable the production of monitoring results for the monitorable properties within this set.

Continuous in the context of adaptive monitoring shall mean that monitoring results can be provided to their consumers without interruption. This also needs to be the case while a runtime model, which is used to access the monitoring results, is evolving in order to support monitoring tasks which were unforeseen at design time. With this paper we are interested in supporting continuous adaptive monitoring through a modeling language which allows runtime model evolution with the same model instance and thus avoiding evolution related interruptions.

3 SCENARIOS

In traditional engineering with design time models and also usually when runtime models are employed, the abstraction respectively reduction is determined at design time. However, in the future systems must be able to learn and adapt to unanticipated changes in the context and therefore also the abstraction respectively reduction present in the runtime models has to change at runtime. For example, machine learning techniques often permit to identify which features are helpful for predictions and therefore future systems require runtime models which (1) can be extended such that new features could be explored and which (2) can be reduced to those features that are in fact really exploited for the predictions. Thus, deciding only at design time about the information that is represented and maintained in the runtime model is not sufficient.

We argue that an evolvable runtime model instance could support continuous monitoring of a running system. The following illustrating scenarios [1] describe exemplarily situations which can occur with a system employing a runtime model. We use them to derive requirements which are later generalized in Section 4 for a modeling language enabling evolvable runtime model instances.

3.1 Running system changes

First we describe the scenarios related to changes of the running system and derive the resulting requirements which need to be supported in combination with continuous adaptive monitoring. To make the scenarios more tangible we consider the ebay-like mRUBiS SEAMS exemplar [16] as the running system.

S1 - System adaptation. The owner of the mRUBiS online auctioning platform wants to ensure that the running system provides a high quality of service and thus utilizes autonomic computing. In this scenario the employed system adaptation engine automatically restarts the malfunctioning query service of the system and increases the value of the corresponding instances pool size parameter. Those temporary changes to the system are also reflected as ordinary changes in the runtime model. Changes in this adaptation scenario do not alter the configuration possibilities of the running system nor what information the runtime model can contain. Resulting requirement: Update system representation structure while the service restarts as well as the pool size parameter value (R1).

S2 - System evolution. The system owner wants to permanently enhance the functionality of the system by providing the users with an additional new payment option. The owner deploys the corresponding add-on component from a third party software manufacturer to the system and wants the component to be represented in the runtime model, too. However, that this particular kind of component would be added to the system one day was not foreseen. Thus, a runtime model classifier for the component needs to be defined belatedly. Resulting requirement: Add a classifier for the new component (R3).

S3 - Software evolution. The software manufacturer is currently working on a new release of the inventory component and wants to perform an experiment with the new version based on real data and user traffic. Thus, some online shop tenants, so called early adopters, get a new version of the inventory component deployed. Now two versions of the inventory component with different sets of properties need to be represented in the runtime model - the old for the normal tenants and the new for the early adopters. Please note that applying the result of software evolution to a system instance, for example, by deploying a new software component version, causes system evolution, too. Resulting requirement: Add a new classifier with the same name but a different version number for the new component release (R3).

S4 - Systems integration and division. In this scenario the system owner is able to expand the business to another region by buying a company there. Instead of operating two systems some parts of the bought system shall be integrated and others be gradually migrated. Both systems have their own runtime model classifier system which shall coexist and stepwise be harmonized. In the meantime the risk of classifier name clashes needs to be precluded. Resulting requirements: Support the two different classifier systems in the runtime model (R7), represent relationships between the parts of the integrated systems (R5), withdraw obsolete classifiers after the corresponding components have been migrated (R4).

3.2 Information demand changes

The scenarios about information demand changes including those regarding the required abstraction level are:

S5 - Filtering. In this scenario the system owner wants to use machine learning to explore the runtime model and identify those monitorable properties per tenant shop, which are good indicators for when to preemptively increase the resources for the bid and buy service to efficiently avoid long response times. For the selected monitorable properties the production of monitoring results shall be continued after the exploration phase. Not required monitoring shall be deactivated afterwards for cost reasons. A generic adaptive monitoring approach based on the actual information demand regarding currently relevant monitoring results has been described in [4]. An approach using machine learning to identify indicators for selecting a system self-healing option is presented in [9]. Resulting requirement: Indicate the actual information demand (R2).

S6 - Aggregation. This scenario covers two different kinds of aggregation which this time also happen at two different places. The first aggregation is called *entity aggregation by function* [8]. In this case the runtime model shall abstract from the individual query component instances, which perform the same functionality in parallel, to one runtime model representation, which allows reasoning about the service which all the instances provide together. This aggregation is performed by the monitoring instruments maintaining the runtime model. Hence, the details of the aggregation are hidden from the runtime model.

The second aggregation is called *entity aggregation by structure* where lower level entities for example integrated components are represented by one higher level entity such as a subsystem [8]. Another difference to the first aggregation is that this time the aggregation takes place in the runtime model and all involved entities shall be present in the model. In this scenario actually an exception counter is introduced which aggregates the counts of all early adopter tenants to alert about its speed of increase.

Please be aware that in general an aggregation can occur on the level of the monitoring instruments or on the level of the runtime model where it is modeled as part of the runtime model. When aggregating on monitoring instruments level then only the aggregation result is represented in the runtime model and its compositional structure remains transparent to runtime model queries. When aggregating on the runtime model level then the compositional structure of the aggregation is modeled and can be queried in the runtime model.

Resulting requirements: Use logical elements and relationships to represent the aggregation results in the runtime model (R8), introduce new classifiers for the aggregation results (R3), withdraw obsolete classifiers for the component instances which are no longer represented in the runtime model (R4), establish relationships to indicate which components the subsystem comprises (R5).

S7 - Itemization. In this scenario the system owner wants to better understand why the user management service sometimes becomes very slow. Thus, the owner introduces additional monitoring capabilities which enable itemizing the service and monitoring its individual subcomponents. The additional data is supposed to help localizing and tackling the response time problem with the service effectively. This scenario could also be combined with the filter scenario (S5). Whenever the response time of the user management service drops below a threshold then the expensive detailed monitoring of its subcomponents could only temporarily be activated. Resulting requirements: Introduce the classifiers of the subcomponents (R3), establish the relationship between the user management component and its subcomponents (R5), optionally activate the monitoring of the subcomponents only when the response time dropped below the threshold (R2).

S8 - Generalization and specialization. As part of the running system each of the online shops can have up to ten different item filters that determine which products are displayed to a shopping user. To indicate potential for configuration optimization the system owner wants to regularly provide a report for each shop about the two filters which filtered the fewest respectively the most items. Instead of having to consider each of the ten filter types individually in a complex runtime model query the system owner prefers a simple query. This query considers all currently deployed filters in a general way not distinguishing between the particular filter types. This is possible because the relevant properties are common to all filter types. To be able to use the simple query the owner introduces a mechanism which ensures that each filter gets assigned to the generic filter classifier in addition to its specific classifier. As a remark, in the case of specialization a more specific classifier would have been assigned. Resulting requirements: Additionally assign the general classifier to each filter (R6).

It needs to be stated that beyond the described scenarios in general filtering, aggregation, itemization, generalization, and specialization shall be possible in the runtime model for structural or value-based monitoring results alike.

4 REQUIREMENTS

In this section we generalize and describe the in Section 3 derived requirements in more detail. For all requirements applies that the corresponding solutions shall allow consumers to access monitoring results through the runtime model continuously. Continuous adaptive monitoring shall be achieved without the need to re-instantiate the runtime model.

R1 - Updating system representation structure and values. The runtime model modeling language and its implementation shall allow updating the structure and values of the system representation programmatically. It shall be possible to add and remove representations of system parts and relationships as well as change the values of other monitorable properties. This requirement only covers changes to the model content representing the running system.

R2 - Indicating the actual information demand. The modeling language and its implementation shall allow recognizing attempts to access monitoring results in the runtime model and processing the corresponding notifications programmatically. This allows the monitoring to adapt the production of monitoring results to the actual demand in a generic way as described in [4].

R3 - Introducing new classifiers including classifier versions. The modeling language and its implementation shall allow defining and afterwards using new classifiers in the runtime model programmatically and without the need to re-instantiate the runtime model. It shall also be possible to programmatically distinguish different versions of a classifier without parsing the classifier name. Thus explicit versioning support is required.

R4 - Withdrawing obsolete classifiers. The modeling language and its implementation shall allow removing classifier definitions which are no longer required. This shall be possible in a programmatic manner and without the need to re-instantiate the runtime model.

R5 - Establishing new kinds of relationships. The modeling language and its implementation shall allow creating relationships with a new classification programmatically and without the need to re-instantiate the runtime model. This means that the decision what relationships of a system part can be represented in the runtime model shall be changeable after the design time.

R6 - Assigning multiple classifiers. The modeling language and its implementation shall allow that system parts and relationships represented in the runtime model can have multiple classifiers. Further it shall be possible to assign those classifiers at different points in time. This shall be possible programmatically and without the need to re-instantiate the runtime model.

R7 - Multiple integrable classifier systems. The modeling language and its implementation shall allow that a runtime model has multiple classifier systems concurrently. Also it shall be possible to link parts which have been classified with different classifier systems and to introduce the classifier systems at different points in time programmatically and without the need to re-instantiate the runtime model. In order to preclude naming conflicts an explicit support for namespaces shall be offered.

R8 - Introducing logical elements and relationships. The modeling language and its implementation shall support logical model elements and relationships which do not physically exist in the running system for example for business related grouping purposes or to store derived information in the runtime model. It shall also be possible to introduce corresponding classifiers programmatically and without the need to re-instantiate the runtime model.

5 APPROACHES

Now we want to investigate to what extent two state-of-the-art approaches support our scenarios for continuous adaptive monitoring and the derived requirements. For this we consider the *classical* model driven engineering approach and an existing runtime model modeling language called *CompArch* [16]. For the classical approach we assume that a runtime model modeling language is defined which is specific to the running system and its domain. This language shall be considered as not intended for reuse with other systems and rather inflexible regarding unforeseeable runtime model changes. In contrast the CompArch language is designed and reused for different scientific experiments with runtime models in the context of self-adaptive systems. Thus, it already provides a certain degree of flexibility. However, as its design goal did not explicitly comprise the support for continuous adaptive monitoring it can also be used to illustrate the need for some additional features. At the end of this section we outline a prospective approach based on a modeling language which is supposed to support the described requirements and scenarios. Table 1 and Table 2 provide an overview by listing the scenarios respectively requirements and indicating how they are supported by the discussed approaches.

Table 1: Supported illustrative scenarios per approach

	Support per approach		
Illustrative scenario	Classical	CompArch	Prospective
S1 - System adaption	✓	✓	✓?
S2 - System evolution	–	(✓)	✓?
S3 - Software evolution	–	(✓)	✓?
S4 - Systems integration and division	–	–	✓?
S5 - Filtering	(✓)	(✓)	✓?
S6 - Aggregation	–	–	✓?
S7 - Itemization	–	–	✓?
S8 - Generalization and specialization	–	–	✓?

✓supported, (✓) partially supported, ✓? shall be supported, – not supported

Table 2: Supported requirements per approach

	Support per approach		
Requirement	Classical	CompArch	Prospective
R1 - Updating system representation structure and values	✓	✓	✓?
R2 - Indicating the actual information demand	(✓)	(✓)	✓?
R3 - Introducing new classifiers including classifier versions	–	(✓)	✓?
R4 - Withdrawing obsolete classifiers	–	✓	✓?
R5 - Establishing new kinds of relationships	–	–	✓?
R6 - Assigning multiple classifiers progressively	–	–	✓?
R7 - Integrating multiple classifier systems	–	–	✓?
R8 - Introducing new logical elements and relationships	–	(✓)	✓?

✓supported, (✓) partially supported, ✓? shall be supported, – not supported

5.1 Classical approach

With the classical approach the decision what kind of information the runtime model can contain, that is which classes of elements and properties, is solely made during design time of the modeling language. The supported classifiers and properties of the runtime model modeling language are defined in the meta-model as shown exemplarily in Figure 1. Using, for example, EMF allows defining the modeling language and generating source code to programmatically create and maintain models based on this language.

Changes to the language imply that the model becomes temporarily unavailable due to the repeated source code generation and the execution of other migration tasks. We want to enable continuous adaptive monitoring through a very long lifetime of runtime model instances. Thus, the classical approach only supports those requirements and consequently scenarios for which it does not require a re-instantiation of the runtime model that means it does not need to be reloaded or recreated due to source code changes of the modeling language implementation. As Table 2 shows, requirement R1 is supported and requirement R2 only partially as the required notification mechanism needs to be explicitly added to the modeling language implementation as described in [4]. The other requirements which demand more flexibility are not supported by this approach. The supported requirements indicated in Table 2 and the resulting requirements listed per scenario in Section 3 allow deriving the illustrative scenarios that are supported by the classical approach. They are also indicated in Table 1.

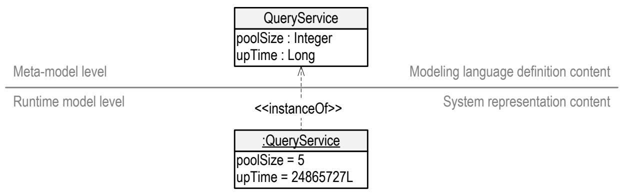

Figure 1: Classical model driven engineering approach - Simple exemplary meta-model and runtime model

5.2 CompArch approach

The classifier system of the EMF-based CompArch approach is split. Software engineering approach specific classifiers such as component and connector are defined in the modeling language meta-model. A small fragment of the CompArch meta-model is depicted in Figure 2. Like with the classical approach all classifiers defined in the meta-model cannot be changed easily. Thus, with CompArch those classifiers are comparatively generic. However, the CompArch meta-model also defines classifiers shown on the left side of the meta-model in Figure 2 which allow the specification of additional more domain-specific classifiers as part of the runtime model classifier definition content, for example, as the query service component classifier is defined in Figure 2. Also following the core of the dynamic object model pattern [11] those specific classifiers can be modeled with an additional model element for each of their properties. A runtime model element of the system representation content can be classified by referencing a classifier defined in the

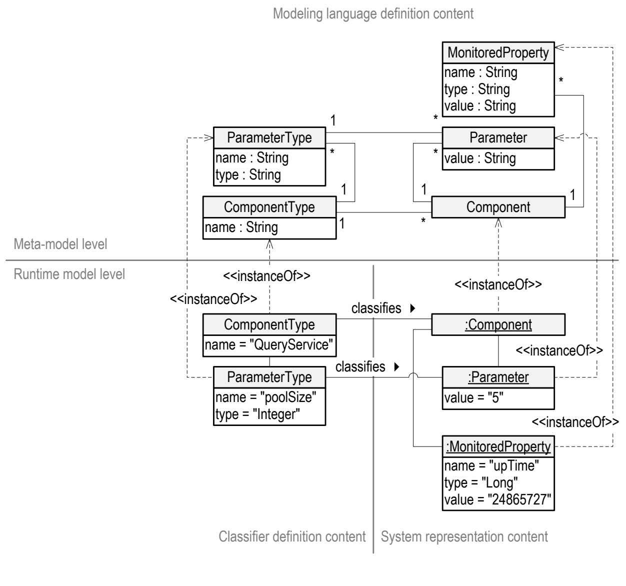

Figure 2: CompArch approach - Simplified meta-model and exemplary runtime model

classifier definition content. This reference needs to be created together with the new model element by a factory, which also needs to create the related parameter and monitored property elements if applicable.

Despite it is not explicitly stated in [16] this allows introducing new classifiers without re-generating source code from the CompArch meta-model and re-instantiating the runtime model.

As Table 2 shows, the requirement R1 is supported. To fully support requirement R2 a minor change to the CompArch implementation is necessary as described in [4] so that the required access notification get sent. Requirement R3 is marked as partially because both, parameters and monitored properties of a component type, should be classifiable by a classifier. Such a parameter or monitored property classifier should also be usable with multiple component types. Also versioning of classifiers is not yet explicitly supported and the approach is specific to components with connectors linking only modeled interfaces. Requirement R4 is supported by deleting the obsolete classifiers form the classifier definition content. The requirements R5 to R7 are not supported due to constraints defined in the CompArch meta-model. Requirement R8 is only partially supported as defining domain-specific classifiers for logical model elements is essentially limited to logical component types. In order to distinguish logical from other elements one needs to apply an according naming scheme. The requirements supported by the CompArch approach as indicated in Table 2 and the resulting requirements listed per scenario in Section 3 lead to the support of the illustrative scenarios as indicated in Table 1.

5.3 Prospective approach

As shown in Table 1 and Table 2 the two investigated state-of-the-art approaches do not cover all the scenarios and requirements which the prospective approach is supposed to support. Like the other two, the prospective approach shall be based on a modeling language. The implementation of this reusable, future language shall allow to programmatically create and maintain software architecture runtime models. Also it shall be flexible enough to enable continuous adaptive monitoring. This means that without the need for new language implementation releases it shall support (1) over time changing information demands regarding the level of abstraction and the monitoring results which are beneficial to produce as well as (2) changes to the modeled running system. This helps to avoid disturbing re-instantiations of the runtime model and thus is a way to enable continuous adaptive monitoring. In order to support the necessary runtime model evolution, the prospective modeling language needs to be generic to a certain degree. However, despite the need for genericness, the language shall still provide enough guidance for creating compatible, in the sense of integrable, runtime models on a consistent and stable conceptual basis. This comprises especially the support for runtime aspects with a likely small set of predefined classifiers in the language meta-model.

6 CONCLUSION AND FUTURE WORK

In this paper we elaborated and described requirements for a prospective modeling language which shall allow creating and maintaining software architecture runtime models for continuous adaptive monitoring. For this purpose we first described illustrative scenarios which can occur with systems employing runtime models. With the scenarios we consider, for example, experimentation in software product development and realizing self-adaptive systems utilizing machine learning. To better understand the resulting requirements and also the implications of not fulfilling them, we investigated to what extent the requirements and the scenarios are already supported by two state-of-the-art modeling language approaches, namely the classical model-driven engineering approach and an existing, reusable, and rather generic modeling language for runtime models. We found that the flexibility which the CompArch modeling language provides by allowing the definition of classifiers in the runtime model is actually particularly helpful to support the stated scenarios. We will continue working towards the prospective runtime model modeling language, for example, by evaluating different implementation options including the use of the dynamic object model pattern. Also we plan to investigate which generic language extensions are beneficial especially regarding the self-adaptation related functions analyze, plan and execute.

REFERENCES

[1] Ian F. Alexander and Ljerka Beus-Dukic. 2009. *Discovering Requirements: How to Specify Products and Services.* Wiley.

[2] Nelly Bencomo, Gordon Blair, Sebastian Götz, Brice Morin, and Bernhard Rumpe. 2013. Report on the 7th International Workshop on Models@Run.Time. *SIGSOFT Software Engineering Notes* (2013).

[3] Amel Bennaceur, Robert France, Giordano Tamburrelli, Thomas Vogel, Pieter J. Mosterman, Walter Cazzola, Fabio M. Costa, Alfonso Pierantonio, Matthias Tichy, Mehmet Akşit, Pär Emmanuelson, Huang Gang, Nikolaos Georgantas, and David Redlich. 2014. Mechanisms for Leveraging Models at Runtime in Self-adaptive Software. In *Models@run.time: Foundations, Applications, and Roadmaps*, Nelly Bencomo, Robert France, Betty H. C. Cheng, and Uwe Aßmann (Eds.).

[4] Thomas Brand and Holger Giese. 2018. Towards Generic Adaptive Monitoring. In *2018 IEEE 12th International Conference on Self-Adaptive and Self-Organizing Systems (SASO).* to appear.

[5] Yuriy Brun, Ron Desmarais, Kurt Geihs, Marin Litoiu, Antonia Lopes, Mary Shaw, and Michael Smit. 2013. A Design Space for Self-Adaptive Systems. In *Software Engineering for Self-Adaptive Systems II: International Seminar, Dagstuhl Castle, Germany, October 24-29, 2010 Revised Selected and Invited Papers*, Rogério de Lemos, Holger Giese, Hausi A. Müller, and Mary Shaw (Eds.).

[6] Frederica Darema. 2004. Dynamic Data Driven Applications Systems: A New Paradigm for Application Simulations and Measurements. In *Computational Science - ICCS 2004.*

[7] Aleksander Fabijan, Pavel Dmitriev, Helena Holmström Olsson, and Jan Bosch. 2017. The Evolution of Continuous Experimentation in Software Product Development: From Data to a Data-Driven Organization at Scale. In *2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE).*

[8] Frederick K. Frantz. 1995. A Taxonomy of Model Abstraction Techniques. In *Proceedings of the 27th Conference on Winter Simulation.*

[9] Sona Ghahremani, Christian M. Adriano, and Holger Giese. 2018. Training Prediction Models for Rule-based Self-adaptive Systems. In *2018 IEEE International Conference on Autonomic Computing (ICAC).* To Appear.

[10] OMG. 2014. Object Management Group Model Driven Architecture (MDA) - MDA Guide rev. 2.0.

[11] Dirk Riehle, Michel Tilman, and Ralph Johnson. 2005. Dynamic Object Model. In *Pattern Languages of Program Design 5*, Dragos Manolescu, Markus Voelter, and James Noble (Eds.).

[12] J. Rothenberg. 1989. The Nature of Modeling. In *Artificial Intelligence, Simulation & Modeling*, Lawrence E. Widman, Kenneth A. Loparo, and Norman R. Nielsen (Eds.).

[13] Dave Steinberg, Frank Budinsky, Marcelo Paternostro, and Ed Merks. 2009. *EMF: Eclipse Modeling Framework.* Addison-Wesley.

[14] Michael Szvetits and Uwe Zdun. 2016. Systematic literature review of the objectives, techniques, kinds, and architectures of models at runtime. *Software & Systems Modeling* (2016).

[15] Thomas Vogel. 2018. *Model-Driven Engineering of Self-Adaptive Software.* Ph.D. Dissertation. University of Potsdam, Germany.

[16] Thomas Vogel. 2018. mRUBiS: An Exemplar for Model-Based Architectural Self-Healing and Self-Optimization. In *International Symposium on Software Engineering for Adaptive and Self-Managing Systems.*