An Expandable Extraction Framework for Architectural Performance Models

Jürgen Walter
University of Würzburg
97074 Würzburg Germany

Christian Stier
FZI Research Center for Information Technology
76131 Karlsruhe Germany

Heiko Koziolek
ABB Corporate Research
68526 Ladenburg Germany

Samuel Kounev
University of Würzburg
97074 Würzburg Germany

ABSTRACT

Providing users with Quality of Service (QoS) guarantees and the prevention of performance problems are challenging tasks for software systems. Architectural performance models can be applied to explore performance properties of a software system at design time and run time. At design time, architectural performance models support reasoning on effects of design decisions. At run time, they enable automatic reconfigurations by reasoning on the effects of changing user behavior. In this paper, we present a framework for the extraction of architectural performance models based on monitoring log files generalizing over the targeted architectural modeling language. Using the presented framework, the creation of a performance model extraction tool for a specific modeling formalism requires only the implementation of a key set of object creation routines specific to the formalism. Our framework integrates them with extraction techniques that apply to many architectural performance models, e.g., resource demand estimation techniques. This lowers the effort to implement performance model extraction tools tremendously through a high level of reuse. We evaluate our framework presenting builders for the Descartes Modeling Language (DML) and the Palladio Component Model (PCM). For the extracted models we compare simulation results with measurements receiving accurate results.

1. INTRODUCTION

During the life-cycle of a software system, performance analysts continuously need to provide answers to and act on performance-relevant questions about response times, resource utilization, bottlenecks, trends, anomalies, etc. It is a common approach to evaluate systems using model-based predictions in addition to measurement-based approaches. Model-based predictions allow for exploration of alternative deployments, architectures, and configurations without the need to test them in a live system. Architectural performance models, a subcategory of quality-aware architecture description languages, can be applied to explore performance properties of a software system for design time and runtime scenarios. At design time models can be applied to reason on effects of design decisions when implementation not fully available yet. At runtime they can be used to reason on effects of changing user behavior on performance to avoid contention via reconfiguration. The main advantage of architectural models, compared to purely predictive models, is that they preserve context information alongside the performance information. Thereby, architectural performance models provide actionable knowledge for automated or manual architectural design decisions. There exist many architectural performance modeling languages, e.g. Descartes Modeling Language (DML) [10], Palladio Component Model (PCM) [4], CACTOS [1], UML MARTE [12], and ACME [17]. Even though the mentioned architectural models focus on different application domain, they have a high semantic overlap. Key concepts such as compositionality, component interfaces, and interface providing roles can be found in all of the mentioned languages. Each of the languages focuses on a different application scenario and is supported by different tool chains for model analysis.

The manual creation of accurate performance models for large scale systems requires extensive effort and knowledge of the architectural modeling language. Existing performance model extraction tooling focuses on the extraction of models for a single language. This requires the reimplementation and maintenance of extraction tooling for each architecture language.

In this work we provide an expandable approach for automated extraction of architectural performance models. Our approach isolates the extraction of shared concepts from language specific implementations. Developers adopting our framework only need to implement a builder interface covering the language specific mapping of common concepts, instead of implementing the entire extraction code. The implementation of our approach, called Performance Model Extractor (PMX), can be reused at different development stages to create performance models of different modeling languages. The remainder of this paper is organized as follows: Section 2 motivates our approach by means of a pro-

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

ICPE '17 Companion, April 22 - 26, 2017, L'Aquila, Italy

© 2017 Copyright held by the owner/author(s). Publication rights licensed to ACM.

ISBN 978-1-4503-4899-7/17/04…$15.00

DOI: http://dx.doi.org/10.1145/3053600.3053634

blem statement. Section 3 presents our approach. Section 4 outlines the builder interface. Section 5 outlines how the framework derives generic information used to trigger the builder. In Section 8 we present the evaluation of our approach. Section 9 concludes.

2. PROBLEM STATEMENT

Architectural performance models can be applied for various purposes and at different development stages [7]. Architectural performance models have been applied for runtime optimization, as well as evolution scenarios. Further, architectural performance models have been applied to evaluate different concerns. Analysis approaches include, e.g., SLA evaluation, or extensions to support energy cost prediction. To enable the application of previously described in an integrated approach, there is very limited tool support. Most analysis techniques are only supported by a single analysis tool chain. An integrated approach would enable the following aspects:

- **Analysis Tool chain Parallelism** "*As a user, I would like to use different tool chains in parallel.*" Advanced Analysis approaches only supported for a limited set of languages. Analysis tool chain parallelism would enable an extended range of analysis techniques.
- **Analysis Toolchain Flexibility** "*As a user, I prefere not to be forced to decide about the toolchain in advance.*" Easy change of modeling formalism.
- **Extraction Toolchain Reuse** "*As a user, I would like to include performance model extraction for newly emerging formalisms without bothering about extraction complexity.*" Currently there is a reinvention of extraction methodology for each modeling formalism. The integration and composition of libraries for the extraction causes huge efforts.

We see a need for an integrated approach that may reuse the same monitoring data for different concerns. However, performance model extraction tooling so far allows only for the extraction of a single language.

3. APPROACH

The goal of our approach is to separate the extraction of shared concepts and concerns found in many architectural performance models, from language specific construction and mapping routines. A key concern shared when creating architectural performance models is to determine resource demands. Resource demands represent the computational demand caused when users issue calls to the services of the modeled system. Resource demand estimation techniques leverage measurement logs to estimate the resource demands of individual service calls [14]. To provide resource demand estimation techniques with sufficient information, the compositional structure and call dependencies between system components need to be extracted. Kieker is an example of monitoring tooling that supports this [20].

In this work we present a framework, called PMX, that provides developers with a solution that integrates established tooling for monitoring [20], log processing [19], and resource demand estimation [14]. To leverage PMX for model construction, developers only have to implement a model builder interface that maps language independent concepts to language specific representations.

PMX employs the builder [8] pattern to decouple language specific mappings from common model extraction concerns. Figure 1 shows the coarse grained architecture including two builder implementations. The intent of the builder design pattern is to separate the construction of a complex composed entity from its representation. Through this, the same construction process can create different representations. The construction complexity of architectural performance models is caused by control flow extraction, the high degree of interconnection between concepts [18] and by the choice of resource demand extraction techniques. By decoupling resource demand estimation and modeling extraction methods for the concepts shared among architectural performance models, PMX reduces the effort for implementing automated performance model extraction.

4. BUILDER INTERFACE

Our approach builds upon the basic assumption that architectural performance modeling languages share equivalence classes of elements. For identification we orient at model-driven software development (MDSD) community. Table 1 shows the core concepts of architectural performance models we identified. They include application architecture concepts like e.g. role, interface, signature, components, service behavior and resource demands. Those concepts occur or correspond with entities in many quality-aware architectural description languages like, e.g., DML, PCM, CACTOS [1], UML MARTE [12], and ACME [17]. The terminology of PMX builds upon the terminology used in DML, PCM and CACTOS. The following outlines the correspondence between the chosen terminology and ADL, if it deviates. For UML Marte (AADL) for Embedded Systems, *ClientServerPorts* with *kind = provided* and *kind = required* correspond to provided and required roles, respectively. *WorkloadBehavior* from the package *PAM_Workload* has the role of a service behavior. In ACME [9], *ports* subsume both required and provided roles of a component. The role of a port in the *connector* between two components identifies a port as either required or provided. The *connectors* correspond with assembly connectors. Service behavior in a set of component *properties* that can be parameterized. The identified core concepts lead to the *builder interface* we present in the following.

```
public interface IModelBuilder {
public EObject createHost(String hostName, int
     numberOfCores);
public EObject createComponent(String componentName);
public EObject createInterface(String
     InterfaceName);
public EObject createMethod(String interfaceName,
     Signature signature);
public EObject createAssembly(String assemblyName,
     String componentName);
public EObject createAllocation(String assemblyName,
     String hostName);
public EObject createProvidedRole(String
     componentName, String interfaceName);
public EObject createRequiredRole(String
     componentName, String interfaceName);
public EObject createServiceBehavior(String
     componentName, String methodName,
     List<ExternalCall> externalCalls, String
     processingResource, double meanResourceDemand);
public void createResourceDemand(String service);
public void createWorkload(HashMap<String,
     List<Double>> workload);
```

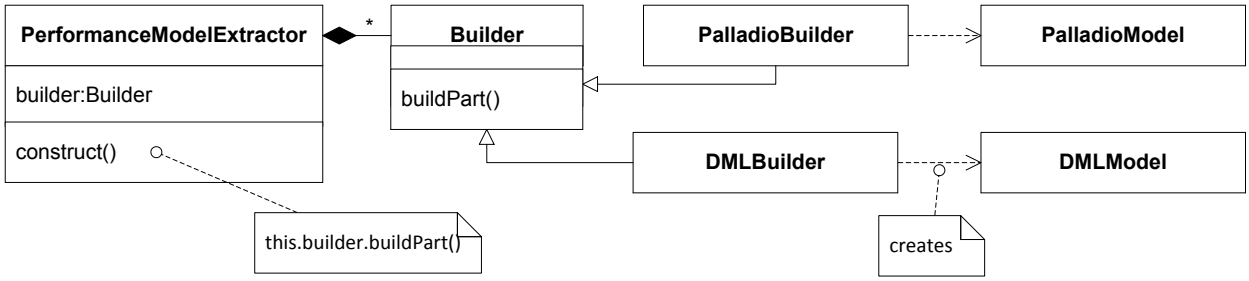

Figure 1: Builder Pattern Architecture of PMX

Concept	Description
Component	Component description includes provided and required roles, and behavior descriptions for the signatures (derived from provided roles).
Interface	contains a set of method signatures
Method signature	operation description that may be equipped with a method parameters and a return value.
Providing and requiring roles	Roles equip component definitions with an external access definition. They contain an interface reference. Mapping between first class entities interface and component.
Assembly	Contexts support the multiple use of the same component type in several environments in an assembly.
Host	Infrastructure element where assembly components can be deployed on.
Assembly connector	After putting components into assembly contexts (from which provided and required context roles can be derived) they can be connected by using system assembly connectors. A system assembly connector connects a required role in of a component in a given assembly context with the provided role of a component in a different assembly context. The referenced provided role and the referenced required role refer to the same interface.
Service behavior	service behavior including resource demands, internal control flow and and external calls. Associated to a component and signature.
Workload	frequency and kind of system requests

Table 1: Core performance anotated architecture description language (ADL) concepts based on [4]

The creation methods of the interface, presented above, represent creation and/or connection functionality. The created objects are referenced by other interface methods during model creation. The implementation of creation methods for host, component, and interface require only object instantiation. The implementation of such creation methods, for a concrete language, is straight forward. Those basic elements are referred to in the performance model composition process at multiple stages. For example, `createMethod` references interface to append signatures, `createProvidedRole`, `createRequiredRole` are appended to existing components and reference the previously created interface, `createResourceDemand` enriches service with internal resource demand. The resource demand is not a parameter as it can be taken from resource demand HashMap using identifier (c.f. Algorithm 1). Other connections include, for example, `connectAssemblies` requires to add references to connected assemblies to the connection element. The function `addComponentToAssembly` sets the component for an assembly.

```
public EObject connectAssemblies(String
     providingAssemblyName, String
     requiringAssemblyName);
public void addComponentToAssembly(String
     assemblyName, String componentName);
```

The model creation process needs to store created elements to access them later for connection and references. Access to previously created elements needs to be done based on identifiers. Hence, the interface also includes various getter functions.

```
public EObject getRole(String role);
public EObject getAssembly(String assemblyName);
public EObject getMethod(String methodName);
public EObject getInterface(String interfaceName);
public EObject getServiceBehavior(String
     componentName,String methodName);
```

To relieve the developer of a builder implementation from implementing all getter functions, we introduced `AbstractModelBuilder` which implements all getter functions of the `IModelBuilder` interface. It stores created elements in hash maps so that they can be referenced within the extraction algorithm. Hence, builder implementations also have to extend `AbstractModelBuilder` to be compatible with the framework.

5. FRAMEWORK IMPLEMENTATION

We build our framework to derive the core aspects of architectural performance models like control flow, resource demands, and workload. Then PMX uses them to trigger the methods of the builder interface to create an architectural performance model. Algorithm 1 represents the performance model extraction receiving the path to monitoring logs and a builder instance. Lines 2-7 extract information

Algorithm 1 Model Extraction Using Generic Builder

```
1: function CONSTRUCT(Path path, Builder builder)
2:     logs ← readLogFiles(path)
3:     analyzer ← compose analysis filters
4:     analyzer.analyze(logs)
5:     operationGraph ← analyzer.getOperationGraph()
6:     rds ← analyzer.getResourceDemands()
7:     workload ← analyzer.getWorkload()
8:     buildModel(operationGraph, rds, workload, builder);
9:     builder.save()
10: end function
```

independent of targeted language description which is used in Line 8 to trigger the `construct` method of the builder pattern that triggers calls to builder methods. The extraction of basic information is based on filters that are connected using pipes-and-filer architecture of Kieker [20]. The processing of monitoring logs is triggered in Line 4. The extraction of control flow is based on Kieker filters that extract operation call graph including calls weights that allow to derive call probabilities (for details see [**?**]). The resource demands extraction filters for method call times and resource logs. After processing of all logs, the framework triggers the *Lib*rary for *Re*source *D*emand *E*stimation (LibReDE) [15, 14] [1] to estimate resource demands. In case resource utilization information is available, estimation is based on service demand law [6]. Otherwise, estimation uses a response time approximation approach. To describe the workload, PMX stores for each interface/role at the system border the arrival times. This allows for the creation of empirical workload models as same as for aggregated probabilistic ones.

Algorithm 2 refines access of builder. Lines 2-5 apply Kiekers `systemModel` containing static system properties (names of hosts, components, interfaces, allocations) which can be received counting divergent identifiers. Each of the named lines iterates over all elements for the type. For example, Line 2 triggers the `createHost` interface method for each host. Interaction element creation happens using call graph processing. Vertices represent methods (including information about component and host). Edges represent calls to other methods. Lines 6-23 process all call graph vertices. Lines 7 and 8 create per edge an assembly and adds component to assembly. method executions. Line 22, uses information that is For each outgoing edge, a service call is created.

6. EXPANDABILITY OF FRAMEWORK

Even our framework covers a full extraction story, it not yet covers all modeling techniques and possible extraction techniques. Our implementation is limited by the builder interface (shared concepts) and by the available input information. In the following we sketch how to expand:

Information retrieval methods To include new information retrieval methods without introducing new elements is possible to extend the framework without changing the builder interface. For example, when the monitoring framework enables to measure resource demands per request, it could be included to replace estimation methods. Further, there exist some limitations using standard monitoring log informations. The model creation of PMX builds upon

[1] http://descartes.tools/librede

Algorithm 2 Application of builder for Performance Model Generation

```
1: function BUILDMODEL(systemModel, operationGraph,
   resourceDemands, workload, builder)
2:     createHosts(systemModel, builder);
3:     createComponents(systemModel, builder);
4:     createInterfaces(systemModel, builder);
5:     createAllocations(systemModel, builder);
6:     for all source : operationGraph.vertices do
7:         component ← source.component.name
8:         host ← source.host.name
9:         assembly ← component + host
10:        builder.addAssembly(assembly);
11:        builder.assign(assembly, component);
12:        for all edge : source.outgoingEdges do
13:            target ← edge.getTargetVertice;
14:            tComponent ← target.component.name
15:            tHost ← target.host.name
16:            tAssembly ← tComponent + tHost
17:            builder.assign(tAssembly, tComponent);
18:            builder.connect(assembly, tAssembly);
19:            calls ← outgoing.getExternalCalls();
20:        end for
21:        rd ← resourceDemands.get(signature)
22:        builder.addBehavior(component, signature,
   calls, host, rd);
23:    end for
24: end function
```

a probabilistic call graph introducing some limitations. For example, it cannot be said whether a loop behavior has been created using a "for" or "while"-operator. Hence, our framework does neither. Moreover, component relations can be extracted while containments cannot be derived from available measurement information. The measurement logs do not uncover whether a method call has been triggered by an interface call or triggered by an event listener. Additional use of source code information could improve the extracted model and retract limitations.

Expanding information sources, additional runtime information, e.g. garbage collection [23] could be included. Some events, like garbage collection, may occur rarely. It depends on chance if such information is included in measurements. In the direction of rare events, the framework could be extended with outlier detection mechanisms.

New features It might be required for some applications to expand PMX to extract additional language features available only in a subset of formalisms. This requires to expand the builder interface. We propose to use template method [8] extending the skeleton of the `CONSTRUCT` method, deferring building again to builders. It is important to provide an empty default implementation to not break other builders and remain downward compatible. Extending the common set of core modeling techniques, there exist concepts which are integrated differently. For example, parametric dependencies (e.g. parametric resource demands and branching probabilities) have been integrated differently. Hence, extensions in this direction should not be required for every builder.

Modeling alternatives Some languages offer to model the same systems properties in various ways. For example, DML offers different granularities (black-box, coarse-

Workload in requests per second	CPU utilization (average)			session response time in ms (average)		
	Actual	DML	PCM	Kieker	DML	PCM
1(calibration)	0.33%	0.35%	0.34%	14.24	14.13	14.13
732	25.22%	24.64%	24.84%	14.35	14.14	14.54
940	33.12%	31.64%	31.77%	15.65	14.14	14.69

Table 2: Evaluation Results Pet Clinic Case Study.

grained, fine-grained) for the behavior description. This can be addressed providing different builder implementations for the same language.

7. STATE OF THE ART

There exist various quality-aware architecture description languages, often focused on performance. A comparison of architectural performance modeling languages has been performed e.g. in [3]. The extraction of such models can be grouped by aspects (e.g. control flow and resource demands), each providing different possibilities for their extraction [21]. For example, the options for the extraction of resource demands include direct measurement and many estimation techniques [24, 14]. In [16] an agent-based model update for online scenarios has been proposed, that updates parts of the model in different frequency and can be applied supplementary to the presented approach. The approach in this papers compares mainly to performance model extraction approaches targeting architectural models. Examples include e.g. [22], [25]. Compared to our contribution, the named approaches are limited to a single modeling language and rely on commercial monitoring infrastructure. In addition to measurement-based extraction, there exist approaches that leverage both static code analysis and measurements to extract architectural performance models [11]. The static analysis allows for the detection of architectural information from source code, e.g., control flow causality for loop counts. However, they require an active profiling of the application to detect these causalities.

8. EVALUATION

To evaluate our framework, we developed two builder implementations of the builder interface. We selected PCM and DML. Both have been applied at design time and runtime scenarios. While the first is more related to design time, the second is more a run time architectural model.

Setting To evaluate our approach, we selected the Pet Clinic application [2] representing a portal for vet appointments. We deployed it on a Dell Power Edge R815 with 48 cores, each core equipped with an Opteron 6174 CPU 2.6 GHz. The application was running on an Ubuntu 14.04.5 VM equipped with 16 GB RAM (to be no bottleneck) and an assignment of 42 cores. We modified the "browse and edit" workload shipped with the application removing the "edit" operations to avoid database contention (due to locking). The resulting workload is open with an exponentially distributed arrival rate. Each vet customer calls the following sequence of interfaces: The vet visits the front page `welcomeGET`, looks at all vets `showVetListGet`, then searches pet owners `initFindFormGet`, `processFindFormGet` and displays all pet owners using `showOwnerGET`. Then the vet triggers two times a specific pet owner page `processFindFormGet`. Besides workload, we modified the application to cache the vet catalog once at startup to improve performance.

The load driver has been deployed at the application machine to avoid a network bottleneck that manifested itself for transaction rates above 400 workload requests per second. We employed a JMeter instance as a load driver and deployed it on a separate VM which received the remaining six cores.

Results For the described setting, we measured a calibration workload at low utilization used to trigger performance model extraction applying the builder implementations for PCM and DML. Then we performed benchmark measurements using the Kieker monitoring framework for different load scenarios and compared them to simulation results of the extracted models. We derived performance metrics in the following way: Kieker log files where filtered using the Descartes Query Language [5] to derive response times. Utilization has been measured using the Linux top command.

DML Analysis used a transformation to Queueing Petri Nets (QPNs) [13] and simulation using SimQPN simulation engine. PCM analysis was performed using SimuLizar [2] simulation engine. Table 2 compares the measured and predicted response times and utilizations. For the evaluated scenarios we receive accurate model-based predictions. The deviation for utilization is below 2% and below 10% for response times.

9. CONCLUSION

In this paper, we present a framework for the extraction of architectural performance models generalizing over the target modeling language. Using the presented approach, the user only has to implement our builder interface to create a performance model generation tool for a specific modeling language. This lowers the effort for such a tool tremendously through a high level of reuse. Our approach enables an easy comparison of architectural performance modeling languages and access to different tool chains. Our evaluation presents accurate prediction results for the extracted models. Source code as well as compiled eclipse plugins for the framework as well as model specific extraction tools (including concrete builder implementations) have been made available online. [3]

Acknowledgments

This work was co-funded by the German Research Foundation (DFG) under grant No. KO 3445/11-1.

[2]https://github.com/spring-projects/spring-petclinic

[3]http://descartes.tools/pmx

References

[1] CACTOS Toolkit Version 2. Technical report, 2016. Last visited on 17.01.2017.

[2] M. Becker, M. Luckey, and S. Becker. Performance analysis of self-adaptive systems for requirements validation at design-time. In *Proceedings of the 9th International ACM Sigsoft Conference on Quality of Software Architectures*, QoSA '13, pages 43–52, New York, NY, USA, 2013. ACM.

[3] S. Becker, L. Grunske, R. Mirandola, and S. Overhage. *Performance Prediction of Component-Based Systems*, pages 169–192. Springer Berlin Heidelberg, Berlin, Heidelberg, 2006.

[4] S. Becker, H. Koziolek, and R. Reussner. The palladio component model for model-driven performance prediction. *J. Syst. Softw.*, 82(1):3–22, Jan. 2009.

[5] M. Blohm, M. Pahlberg, S. Vogel, J. Walter, and D. Okanovic. Kieker4DQL: Declarative Performance Measurement. In *Proceedings of the 2016 Symposium on Software Performance (SSP)*, November 2016.

[6] F. Brosig, N. Huber, and S. Kounev. Automated extraction of architecture-level performance models of distributed component-based systems. In *Proceedings of the 26th IEEE/ACM International Conference on Automated Software Engineering (ASE 2011)*, pages 183–192, 2011.

[7] A. Brunnert, A. van Hoorn, F. Willnecker, A. Danciu, W. Hasselbring, C. Heger, N. Herbst, P. Jamshidi, R. Jung, J. von Kistowski, A. Koziolek, J. Kroß, S. Spinner, C. Vögele, J. Walter, and A. Wert. Performance-oriented DevOps: A research agenda. Technical Report SPEC-RG-2015-01, SPEC Research Group — DevOps Performance Working Group, Standard Performance Evaluation Corporation (SPEC), 2015.

[8] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-oriented Software*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1995.

[9] D. Garlan, R. T. Monroe, and D. Wile. Foundations of component-based systems. chapter Acme: Architectural Description of Component-based Systems, pages 47–67. Cambridge University Press, New York, NY, USA, 2000.

[10] S. Kounev, N. Huber, F. Brosig, and X. Zhu. A Model-Based Approach to Designing Self-Aware IT Systems and Infrastructures. *IEEE Computer*, 49(7):53–61, July 2016.

[11] K. Krogmann, M. Kuperberg, and R. Reussner. Using genetic search for reverse engineering of parametric behavior models for performance prediction. *IEEE Transactions on Software Engineering*, 36(6):865–877, Nov 2010.

[12] F. Mallet, C. André, and J. DeAntoni. Executing aadl models with uml/marte. In *2009 14th IEEE International Conference on Engineering of Complex Computer Systems*, pages 371–376, June 2009.

[13] P. Meier, S. Kounev, and H. Koziolek. Automated transformation of component-based software architecture models to Queueing Petri Nets. In *19th IEEE/ACM Int. Symposium on Modeling, Analysis and Simulation of Computer and Telecommunication Systems*, 2011.

[14] S. Spinner, G. Casale, F. Brosig, and S. Kounev. Evaluating Approaches to Resource Demand Estimation. *Performance Evaluation*, 92:51 – 71, October 2015.

[15] S. Spinner, G. Casale, X. Zhu, and S. Kounev. LibReDE: A Library for Resource Demand Estimation. In *Proceedings of the 5th ACM/SPEC International Conference on Performance Engineering (ICPE 2014)*, pages 227–228, New York, NY, USA, March 2014. ACM Press.

[16] S. Spinner, J. Walter, and S. Kounev. A Reference Architecture for Online Performance Model Extraction in Virtualized Environments. In *Proceedings of the 2016 Workshop on Challenges in Performance Methods for Software Development (WOSP-C'16) co-located with 7th ACM/SPEC International Conference on Performance Engineering (ICPE 2016)*, March 2016.

[17] B. Spitznagel and D. Garlan. Architecture-based performance analysis. In *Proceedings of the 1998 Conference on Software Engineering and Knowledge Engineering*, pages 146–151, 1998.

[18] M. Strittmatter, G. Hinkel, M. Langhammer, R. Jung, and R. Heinrich. Challenges in the evolution of metamodels: Smells and anti-patterns of a historically-grown metamodel. In *10th International Workshop on Models and Evolution (ME)*. CEUR Vol-1706, October 2016.

[19] A. van Hoorn. *Model-Driven Online Capacity Management for Component-Based Software Systems*. Number 2014/6 in Kiel Computer Science Series. Department of Computer Science, Kiel University, Kiel, Germany, 2014. Dissertation, Faculty of Engineering, Kiel University.

[20] A. van Hoorn, J. Waller, and W. Hasselbring. Kieker: A framework for application performance monitoring and dynamic software analysis. In *3rd ACM/SPEC Int. Conf. on Perf. Eng. (ICPE '12)*, pages 247–248, 2012.

[21] J. Walter, A. D. Marco, S. Spinner, P. Inverardi, and S. Kounev. Online Learning of Run-time Models for Performance and Resource Management in Data Centers. In S. Kounev, J. O. Kephart, A. Milenkoski, and X. Zhu, editors, *Self-Aware Computing Systems*. Springer Verlag, Berlin Heidelberg, Germany, 2017.

[22] F. Willnecker, A. Brunnert, W. Gottesheim, and H. Krcmar. Using dynatrace monitoring data for generating performance models of java ee applications. In *Proceedings of the 6th ACM/SPEC International Conference on Performance Engineering*, ICPE '15, pages 103–104, New York, NY, USA, 2015. ACM.

[23] F. Willnecker, A. Brunnert, B. Koch-Kemper, and H. Krcmar. Full-stack performance model evaluation using probabilistic garbage collection simulation. In *Proceedings of the 2015 Symposium on Software Performance (SSP 2015)*, 2015.

[24] F. Willnecker, M. Dlugi, A. Brunnert, S. Spinner, S. Kounev, and H. Krcmar. Comparing the Accuracy of Resource Demand Measurement and Estimation Techniques. In M. Beltrán, W. Knottenbelt, and J. Bradley, editors, *Computer Performance Engineering - Proceedings of the 12th European Workshop (EPEW 2015)*, volume 9272 of *Lecture Notes in Computer Science*, pages 115–129. Springer, August 2015.

[25] F. Willnecker and H. Krcmar. Optimization of deployment topologies for distributed enterprise applications. In *2016 12th International ACM SIGSOFT Conference on Quality of Software Architectures (QoSA)*, pages 106–115, April 2016.