Performance Queries for Architecture-Level Performance Models*

Fabian Gorsler
Karlsruhe Institute of Technology (KIT)
Am Fasanengarten 5
76131 Karlsruhe, Germany
gorsler@ira.uka.de

Fabian Brosig
Karlsruhe Institute of Technology (KIT)
Am Fasanengarten 5
76131 Karlsruhe, Germany
brosig@kit.edu

Samuel Kounev
Karlsruhe Institute of Technology (KIT)
Am Fasanengarten 5
76131 Karlsruhe, Germany
kounev@kit.edu

ABSTRACT

Over the past few decades, many performance modeling formalisms and prediction techniques for software architectures have been developed in the performance engineering community. However, using a performance model to predict the performance of a software system normally requires extensive experience with the respective modeling formalism and involves a number of complex and time consuming manual steps. In this paper, we propose a generic declarative interface to performance prediction techniques to simplify and automate the process of using architecture-level software performance models for performance analysis. The proposed Descartes Query Language (DQL) is a language to express the demanded performance metrics for prediction as well as the goals and constraints in the specific prediction scenario. It reduces the manual effort and the learning curve when working with performance models by a unified interface independent of the employed modeling formalism. We evaluate the applicability and benefits of the proposed approach in the context of several representative case studies.

Categories and Subject Descriptors

D.2.8 [**Software Engineering**]: Metrics—*performance measures*; D.3.2 [**Programming Languages**]: Language Classifications—*specialized application languages*

General Terms

Performance, Languages

Keywords

Software Performance Engineering, Performance Prediction, Automation, Query Language, Domain-specific Language

*This work was funded by the German Research Foundation (DFG) under grant No. KO 34456-1.

ICPE'14, March 22–26, 2014, Dublin, Ireland.

http://dx.doi.org/10.1145/2568088.2568100.

1. INTRODUCTION

Performance and availability are crucial for today's software systems [22, 33]. Modern IT solutions introduce additional abstraction layers such as virtualization layers and need to sustain increasing workloads. The increasing complexity makes providing adequate performance a challenging task. Analyzing the performance characteristics of a software system during all phases of its lifecycle helps to avoid performance problems.

The performance of a software system can normally be analyzed through performance prediction techniques based on performance modeling formalisms. These techniques and formalisms differ in their expressiveness, prediction capabilities, computing effort and modeling effort. We distinguish between two major families of performance models. *Predictive performance models* such as Queueing Networks (QNs), Queueing Petri Nets (QPNs) or Layered Queueing Networks (LQNs) focus on capturing the temporal system behavior. They are used on a high level of abstraction and can be solved analytically or by simulation techniques [22, 17, 21]. *Architecture-level performance models* describe the software architecture, the deployment and are annotated with performance-relevant aspects of the software system. Prominent examples are the UML SPT profile [23] and its successor the UML MARTE profile [24], CSM [38], PCM [2] and KLAPER [10]. To predict performance metrics, automated model-to-model transformations into predictive performance models are normally employed.

During the performance prediction process, users execute or trigger the following tasks as shown in Fig. 1. When a performance analysis is triggered, the architecture-level performance model is transformed into a suitable predictive performance model. The resulting model is then solved by analytical or simulation means to derive performance metrics. Finally, the user extracts the metrics of interest. However, the presented process has several shortcomings arising from a lack of automation and required manual efforts: While the transformation and model solving steps are typically automated, their configuration has to be done manually and is dependent on the tooling and the performance model formalism. Furthermore, depending on the output of the model solving step, the extraction of the performance metrics of interest is a manual step and requires performance modeling expertise of the user.

In this paper, we propose a generic declarative interface for performance prediction techniques to simplify and automate the process of using architecture-level software per-

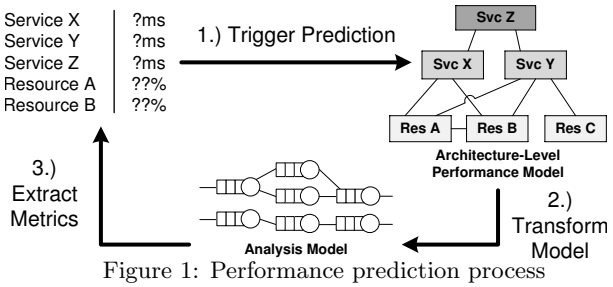

Figure 1: Performance prediction process

formance models for performance analysis. The interface allows the formulation of performance queries. A performance query specifies which performance metrics should be predicted under which scenario-specific constraints. Furthermore, performance queries may involve parameter variations for different performance model parameters such as request arrival rates or thread pool sizes.

The unified interface that is independent of the employed modeling formalism reduces the manual effort and learning curve when working with performance models. We provide our approach as an extensible architecture with an implementation that is capable to integrate third-party extensions supporting specific performance modeling formalisms and prediction techniques. To demonstrate the applicability and benefits of our approach, we present representative case studies showing how to integrate different established performance prediction techniques. Other tool developers can leverage our approach by offering an interface for their performance modeling formalism and prediction technique.

In summary, the contributions of this paper are: (i) the Descartes Query Language (DQL), a novel query language to specify performance queries with optional parameter variations, (ii) an architecture and implementation that unifies the prediction process by using DQL as declarative interface, and (iii) a detailed evaluation of the applicability and benefits of our approach in the context of several representative case studies.

The remainder of the paper is organized as follows: First, a requirements specification based on usage scenarios and user stories is presented in Sec. 2. Then we describe the DQL approach in Sec. 3. We show syntax diagrams of DQL and explain its different features. In Sec. 4, we present the architecture and implementation of our corresponding toolchain. Sec. 5 provides an overview of related work. In Sec. 6, we evaluate the applicability and benefits of the proposed approach in the context of several representative case studies. Sec. 7 concludes the paper and provides an outlook on future work.

2. REQUIREMENTS

We investigate common usage scenarios in performance engineering to derive user stories as requirements for our language for performance queries. The presented usage scenarios are based on examples from literature [3, 29, 34]. They vary in the user type and the user role. Furthermore, we distinguish whether the performance queries are issued in an *online* or *offline* context, i.e., if the system is at runtime during operation or if the system is in the development or deployment phase.

2.1 Usage Scenarios

Design Time. At software design time, a software architect tries to find a suitable assembly of components to build a software system. Using an architecture-level performance model of the system, the software architect can simulate different assemblies and configurations to predict the performance behavior. The predictions are not constrained to complete within strict time bounds, but should allow qualitative comparisons of design alternatives with high accuracy. Furthermore, the software architect may want to optimize compositions or configuration parameter settings. For that purpose, an automated design and parameter space exploration covering defined Degrees of Freedom (DoFs) is helpful [19, 14]. DoFs specify how entities in a performance model can be varied and thus span the space of valid configurations and parameter settings. Depending on the size of the configuration space and the space exploration strategy, the time-to-result of a single performance prediction gains in importance. Otherwise, the exploration might take too long to be feasible.

Deployment Time. At deployment time, a software deployer tries to size the resource environment so that the system on the one hand satisfies performance objectives and on the other hand does not waste resources. Using a performance model, this system sizing and capacity planning step can be facilitated. Expensive performance tests can be avoided, because different load situations can be simulated on different resource settings.

System Run-Time. A proactive online performance and resource management aims at adapting system configuration and resource allocations dynamically. Overload situations should be anticipated and suitable reconfigurations should be triggered *before* Service Level Agreements (SLAs) are violated. Performance predictions need to be conducted to answer questions such as: What performance would a new service or application deployed on the infrastructure exhibit and how much resources should be allocated to it? How should the workloads of the new service/application and existing services be partitioned among the available resources so that performance requirements are satisfied and resources are utilized efficiently? What would be the performance impact of adding a new component or upgrading an existing component as services and applications evolve? If an application experiences a load spike or a change of its workload profile, how would this affect the system performance? Which parts of the system architecture would require additional resources? What would be the effect of migrating a service or an application component from one physical server to another? However, there is a trade-off between prediction accuracy and time-to-result. There are situations where the prediction results need to be available very fast to adapt the system *before* SLAs are violated. An accurate fine-grained performance prediction comes at the cost of a higher prediction overhead and longer prediction durations. Coarse-grained performance predictions allow speeding up the prediction process. This trade-off should be configurable when conducting performance predictions.

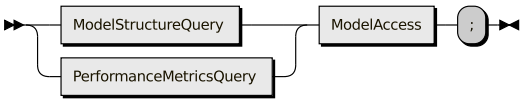

Figure 2: Descartes Query Language (DQL)

2.2 User Stories

We formulate the following user stories as requirements for the query language.

- As a user, I want to issue queries independent of the underlying performance modeling formalism.
- As a user, I want to list the modeled services of a selected performance model instance.
- As a user, I want to list the modeled resources of a selected performance model instance.
- As a user, I want to list the variable parameters of a selected performance model instance.
- As a user, I want to list supported performance metrics for selected services and resources.
- As a user, I want to conduct a prediction of selected performance metrics of selected services and resources.
- As a user, I want to aggregate retrieved performance metrics by statistical means.
- As a user, I want to control the performance prediction by specifying a trade-off between prediction speed and accuracy.
- As a user, I want to conduct a sensitivity analysis for selected parameters in defined parameter spaces.
- As a user, I want to query revisions of model instances.

We emphasize that the query language needs to be independent of a specific performance modeling formalism. Predictable performance metrics may vary from model instance to model instance and model solver to model solver, so the query mechanism needs to include means to evaluate the underlying model and list queryable performance metrics to the user. For each performance modeling formalism, the query language requires a *Connector* to bridge the mentioned gaps. Furthermore, queries should be user-friendly to write, i.e., there should be a text editor with syntax highlighting and auto-completion features.

3. QUERY LANGUAGE

In this section, we present the concepts of the performance query language. We use syntax diagrams to describe the most relevant parts of the language grammar. Details can be found in [9]. Fig. 2 shows the uppermost grammar rule. In general, there are two query classes: (i) A *ModelStructureQuery* is used to analyze the structure of performance models. It can provide information about available services and resources, performance metrics as well as model variation points. (ii) A *PerformanceMetricsQuery* is used to trigger actual performance predictions. Both query classes are followed by a *ModelAccess* part that refers to a performance model instance.

3.1 Model Access

The query language is independent of a specific performance modeling formalism. Thus, to issue a query on a performance model instance, both the location of the model instance as well as a *DQL Connector* need to be specified. A

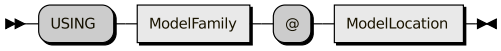

Figure 3: Model Access

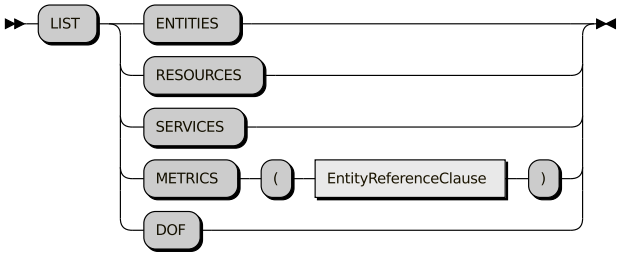

Figure 4: Model Structure Query

DQL Connector is specific for a performance modeling formalism and bridges the gap between performance model and DQL. Fig. 3 shows the model access initiated by keyword `USING`. The nonterminal *ModelFamily* refers to an identifier that serves as reference to a DQL Connector. The DQL Connector has to be registered in a central DQL Connector registry (see Section 4). The *ModelLocation* is a reference to a model instance location.

3.2 Model Structure Query

The user can request information about which services or resources are modeled, and for which model entities the referred model instance can provide which performance metrics. In DQL notation, a model entity is either a resource or a service. Fig. 4 shows a *ModelStructureQuery* initiated by keyword `LIST`. Using terminals `ENTITIES`, `RESOURCES`, `SERVICES` the user can query for respective entities, resources or services. The result is a list of entity identifiers that are unique for the referred model instance. Listing 1 illustrates a simple query example.

Besides querying for services and resources, the user can also query for available performance metrics. We denote a performance metric as *available* if the performance metric can be derived from the performance model instance. For example, for a Central Processing Unit (CPU) resource of an application server, the average utilization is typically available. Since the available performance metrics may differ from entity to entity, the user has to specify for which entity the available performance metrics should be listed. For that purpose, the user has to provide an *EntityReferenceClause*. A *EntityReferenceClause* is a comma-separated list of *EntityReference*s whose syntax is illustrated in Fig. 5.

An entity reference thus starts with a keyword identifying the entity type (`RESOURCE` or `SERVICE`) followed by an entity identifier and an optional *AliasClause*. Listing 2 shows a corresponding query example. In the example, the user queries for available metrics of a resource with identifier 'AppServer-CPU1' and a service with identifier 'newOrder'. For the re-

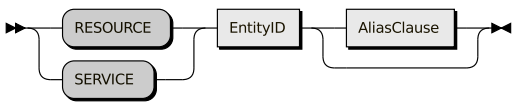

Figure 5: Entity Reference

```
LIST ENTITIES
USING connector@location;
```

Listing 1: List all Modeled Entities

```
LIST METRICS
 (RESOURCE 'AppServerCPU1' AS r,
  SERVICE 'newOrder' AS s)
USING connector@location;
```

Listing 2: List available Performance Metrics

source and the service the user sets alias `r` respectively alias `s`.

Furthermore, DQL allows querying for model variation points, also denoted as Degrees of Freedom (DoFs). The query result then is a list of DoF identifiers. The way how model variation points are modeled is independent from DQL, it depends on the DQL Connector. We provide an example of DoF queries in Section 3.3.2.

3.3 Performance Metrics Query

A *PerformanceMetricsQuery* is used to trigger performance predictions. Fig. 6 shows the syntax. First, we explain the parts of the query that are obligatory to write basic queries. For optional extensions such as query constraints, evaluations of DoFs and model revisions, we refer to Subsections 3.3.1, 3.3.2 and 3.3.3.

A user can specify the performance metrics of interest with wildcard '*' (all available performance metrics) or via the nonterminal *MetricReferenceClause*. *MetricReferenceClause* is a comma-separated list of *MetricReference*s that is shown in Fig. 7. A *MetricReference* either refers to a single metric or to an aggregated metric. A single metric is described by an *EntityIdOrAlias* followed by a dot and a *MetricId* or wildcard. Listing 3 shows a corresponding example where the utilization of an application server CPU and the average response time of a service is requested.

A specification of an aggregated metric consists of two parts. The first part (nonterminal *AggregateFunction*) selects an aggregate function. The set of supported aggregate functions is based on the descriptive statistics part of Apache Commons Math[1] and provides common statistical means, e.g. arithmetic and geometric mean, percentiles, sum and variance. The second part describes the list of performance metrics that should be aggregated. A wildcard ('*') can be used to iterate over all entities where a specific performance metric is available. An exemplary use of an aggregated metric is shown in Listing 4, where the mean value of two application server utilization rates is computed. Note that the computation of the aggregate is provided by the DQL Query Execution Engine (QEE) (see Section 4) and *not* part of a DQL Connector. The DQL Connector only needs to support querying single metrics.

[1] `http://commons.apache.org/proper/commons-math/`

```
SELECT r.utilization, s.avgResponseTime
FOR RESOURCE 'AppServerCPU1' AS r,
    SERVICE 'newOrder' AS s
USING connector@location;
```

Listing 3: Trigger Basic Performance Prediction

```
SELECT MEAN(r1.utilization, r2.utilization)
FOR RESOURCE 'AppServer1' AS r1,
    RESOURCE 'AppServer2' AS r2
USING connector@location;
```

Listing 4: Query with Aggregated Metric

Figure 8: Constraint Clause

3.3.1 Constraints

In online performance and resource management scenarios, controlling performance predictions by specifying a trade-off between prediction accuracy and time-to-result can be important to act in time [16]. DQL allows the specification of such constrained performance queries. The syntax of the corresponding *ConstraintClause* is shown in Fig. 8. The *ConstraintID*s are DQL Connector specific and are intended to control the behavior of the underlying model solving process. For instance, a DQL Connector might support a constraint named `'FastResponse'` to trigger fast analytical mean-value solvers or a constraint named `'Detailed'` to trigger a full-blown simulation that may take a significant amount of time but is able to simulate, e.g., fine-grained OS-specific scheduling behavior [11]. Listing 5 shows a corresponding example.

3.3.2 Degrees of Freedom (DoF)

To evaluate DoFs, DQL provides several optional language constructs. Fig. 9 shows the syntax diagram of nonterminal *DoFClause*. A *DoFClause* refers to DoFs (nonterminal *DoFReferenceClause*) and an optional exploration strategy.

A *DoFReferenceClause* is a comma-separated list of nonterminal *DoFReference* that is shown in Fig. 10. It starts with a DoF identifier (with an optional alias) and is followed by *DoFVariationClause* that provides optional parameter settings (see Fig. 12). In its current version, DQL supports lists of parameter values of type `Integer` or `Double` as well as interval definitions of type `Integer`. Listing 6 shows an example with two DoFs. On the one hand, we vary the inter-arrival time of the open workload (values 0.1, 0.2, 0.3), on the other hand we vary the size of the database connec-

```
SELECT r.utilization, s.avgResponseTime
CONSTRAINED AS 'FastResponse'
FOR RESOURCE 'AppServerCPU1' AS r,
    SERVICE 'newOrder' AS s
USING connector@location;
```

Listing 5: Constrained Query

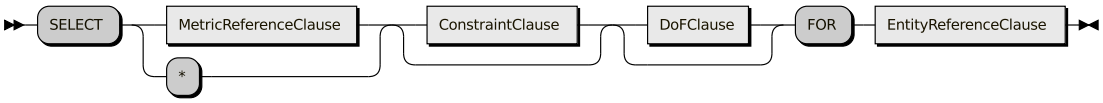

Figure 6: Performance Metrics Query

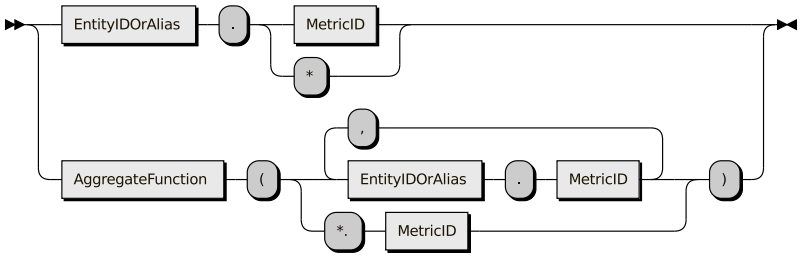

Figure 7: Metric Reference

Figure 10: DoF Reference

Figure 12: DoF Configuration Property

```
SELECT r.utilization, s.avgResponseTime
EVALUATE DOF
  VARYING
    'DoF_OpenWorkload_InterarrivalTime'
      AS dof1 <0.1, 0.2, 0.3>,
    'DoF_JDBCConnectionPool_Size'
      AS dof2 <10..30 BY 5>
FOR RESOURCE 'AppServerCPU1' AS r,
    SERVICE 'newOrder' AS s
USING connector@location;
```

Listing 6: DoF Query

```
SELECT s.avgResponseTime
EVALUATE DOF
  VARYING 'DoF_AppServerVM_Migration' AS dof1
  GUIDED BY 'MyExplorationStrategy'
    [dof1.targets =
     'PhysicalMachineA,PhysicalMachineB']
FOR SERVICE 'newOrder' AS s
USING connector@location;
```

Listing 7: DoF Query with Exploration Strategy

tion pool from 10 to 30 in steps of 5. Without an explicitly defined exploration strategy, the default exploration strategy is considered to be a full exploration. In the example, this means that $3 \times 5 = 15$ performance predictions are triggered. The query result set is then a list of 15 prediction results. Each prediction result contains the prediction for performance metrics `r.utilization` and `s.avgResponseTime`.

Using the optional *ExplorationStrategyID*, together with user-defined configuration properties (see Fig. 12), it is possible to trigger an alternative exploration strategy provided that the DQL Connector supports it. This is necessary for, e.g., DoFs representing migrations of Virtual Machines (VMs) from a physical host machine to another physical host machine. In these cases, it is the DQL Connector that needs to provide means to iterate the configuration space. An integration of complex exploration strategies, e.g., multi-attribute Quality of Service (QoS) optimization techniques to derive Pareto-optimal solutions [18], is thus supported. Listing 7 shows a query example with an explicit exploration strategy. The query has one DoF, namely the physical machine where the appserver VM is assigned to. The configuration space, a set of two physical machines, is described as String as value of property `targets`. Note that the semantics of the configuration properties are specific for the DoF and the exploration strategy, i.e., the properties are not interpreted by DQL.

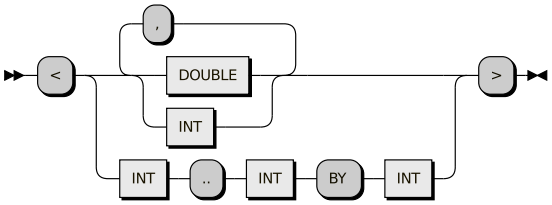

Figure 11: DoF Variation Clause

3.3.3 Temporal Dimension

As additional feature for Performance Metrics Queries, DQL offers facilities to access different revisions of a performance model. The assumption is that the model instances are annotated with a revision number and/or timestamp, i.e., that there is a chronological order.

In particular if the performance models are used in online scenarios, queries that allow the user to ask about performance metrics in the past, the development of performance metrics over time, or (together with a workload forecasting mechanism [12]) the development of performance metrics in the next time, are desirable. In online scenarios, performance model instances are typically part of a *performance*

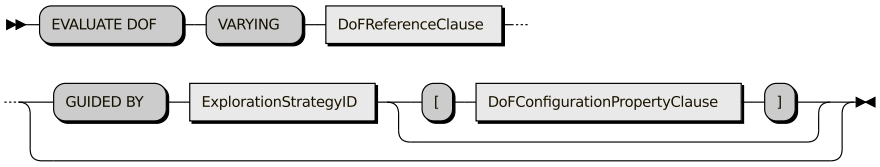

Figure 9: DoF Clause

```
SELECT r.utilization, s.avgResponseTime
FOR RESOURCE 'AppServerCPU1' AS r,
    SERVICE 'newOrder' AS s
USING connector@location
OBSERVE
  BETWEEN '2013-10-09 08:00:00'
   AND '2013-10-10 08:00:00'
  SAMPLED BY 1h;
```

Listing 8: Performance Metrics Over Time

```
SELECT r.utilization
FOR RESOURCE 'AppServerCPU1' AS r
USING connector@location
OBSERVE
  NEXT 2h SAMPLED BY 10M;
```

Listing 9: Anticipated Resource Utilization

data repository that integrates revisions of calibrated model instances and performance monitoring data [16].

DQL allows to express the temporal dimension in different ways: (i) with a time frame defined by a start and end, and (ii) with a time frame starting or ending with the current time and a time delta. Alternative (i) is used in the example in Listing 8. Resource utilization and service response time are queried for a specific time frame of one day. The results are sampled groups of one hour, possibly read from historical monitoring data, thus leading to a set of 24 result sets. The example shown in Listing 9 uses alternative (ii). The query requests the application server CPU utilization for the next two hours, sampled in twelve groups of ten minutes length each. This query triggers performance predictions, provided that a workload forecast for the next two hours is available.

4. TECHNICAL REALIZATION

Based on our query language as described in the previous sections, we provide an implementation of DQL based on an extensible software architecture. The DQL environment is built up on the OSGi Framework, Eclipse Modeling Framework (EMF) and Xtext[2]. In the following we present our realization of the DQL environment, describe internal data structures, outline control flows and give an outline for contributions of custom DQL Connectors.

4.1 Architecture and Components

The architecture of the DQL approach is shown in Fig. 13. It is realized in Java and based on the OSGi Framework [26, 27]. The current implementation is based on plain OSGi operations and runs on Eclipse Equinox[3]. Each component is encapsulated in a dedicated OSGi Bundle and activated on demand. For the interaction among Bundles, the *OSGi Service Layer* is utilized, which results in an event-driven interaction of components triggered by incoming queries.

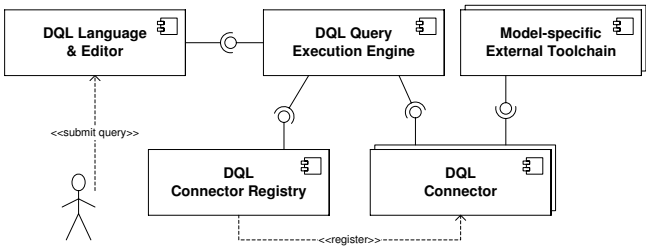

Figure 13: DQL System Architecture

The first component, *DQL Language & Editor*, provides the interface to users and offers an Application Programming Interface (API). The component is based on Xtext. Xtext is a framework to develop Domain-specific Languages (DSLs) and offers facilities to generate software artifacts such as text editor and parser based on a grammar specification. The component provides a DQL query parser and represents statements in an EMF model of the abstract syntax tree. For convenience, an Eclipse-based editor is also part of DQL. The Xtext-generated editor is customized, e.g., code assistance to obtain identifiers of model entities or available performance metrics. Furthermore, users can issue queries and visualize query results.

The second component, *DQL Query Execution Engine (QEE)*, provides the main execution logic in the DQL system architecture. Here, all tasks take place that are independent of a specific performance modeling formalism and prediction technique. The DQL QEE transforms queries into the internal abstraction for queries, the *Mapping Meta-Model*, which will be introduced in Sec. 4.2. The DQL QEE then selects an adequate DQL Connector to access the requested model instance, to execute the query, and to provide the results to the user. The DQL QEE also calculates aggregate functions if requested and performs the necessary pre- and post-processing steps.

The third component, a *DQL Connector*, provides functionality that is dependent on a specific performance modeling formalism and prediction technique. This includes accessing performance models, triggering the prediction process and providing additional information, e.g., about the model structure and available performance metrics. To integrate different approaches for performance prediction in a DQL environment, multiple DQL Connectors can be deployed to the OSGi run-time. As each DQL Connector

[2] http://www.eclipse.org/Xtext/

[3] http://www.eclipse.org/equinox/

comes with a unique identifier that needs to be referenced in a DQL query together with a model location, the DQL QEE can select a suitable DQL Connector to execute the query. To find a suitable DQL Connector, the *DQL Connector Registry* is used. The latter uses the *OSGi Declarative Service* interface to build and maintain an index of available DQL Connectors and their support for query classes.

Our proposed architecture allows to integrate various different performance modeling approaches and prediction techniques. Extensions of the DQL environment are primarily possible by contributing additional DQL Connectors. DQL Connector implementations can be partially, which allows to implement approaches that are not capable of all features of DQL.

4.2 Mapping Meta-Model

The Mapping Meta-Model is an abstraction layer to encapsulate different performance modeling formalisms, prediction approaches and requests resulting from DQL queries. An instance of the Mapping Meta-Model is used to (i) send a DQL query request from DQL QEE to a DQL Connector, to (ii) send the query result back from the DQL Connector to DQL QEE, and (iii) to present the query result to the user.

Fig. 14 shows the Mapping Meta-Model. `EntityMapping` is the top-level type and stores references to different parts of an architecture-level performance model or a related performance modeling formalism. `Resource` and `Service` reference elements of a performance model instance by an identifier. Both are derived from `Entity` which represents any kind of a performance-relevant model entity with an absolute identifier. `Probe` is attached to type `Entity` and either requests the prediction of a specific performance metric or, if the query has been processed, represents the value of the requested metric in an instance of a derived type of type `Result`, e.g., type `DecimalResult`. Thus, in case a Mapping Meta-Model instance contains instances of `Probe`, it is a request to be processed by a DQL Connector, otherwise it represents a result of a query. The type `DoF` represents the usage of a DoF in a query, or, if the query has been processed, represents a specific parameter setting of a DoF during a performance prediction. The type `Aggregate` is inserted in the post-processing step of DQL QEE and represents an aggregate computed on top of returned performance metric values. An example for the usage of the Mapping Meta-Model follows in the next section, detailed usage scenarios and exemplary instances are shown in [9, p. 52 ff.].

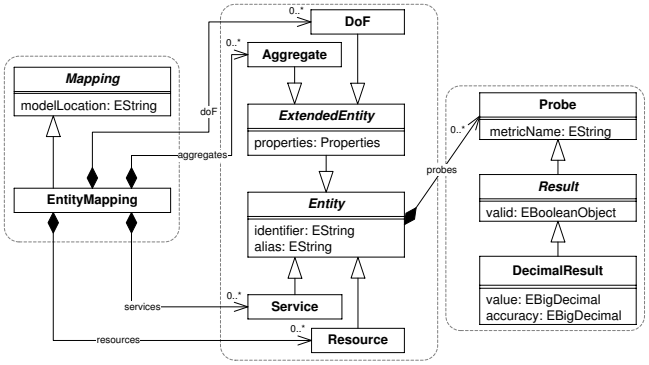

Figure 14: Mapping Meta-Model

4.3 Query Execution

This section describes the sequence of query processing steps of Performance Metrics Queries. The process shown here focuses on the interaction to trigger a performance prediction and to return performance metrics to a user. Fig. 15 shows an overview of the execution sequence that consists of phases that are independent of a modeling formalism and phases that are modeling formalism specific, involving both the DQL QEE and a DQL Connector.

In the first phase, processing takes place at the DQL QEE. The steps are independent of a performance modeling formalism. They involve the look-up of a DQL Connector and its invocation, operations to parse the query, preparation of the request and the setting of configuration options for the DQL Connector. The relevant parts of the query are

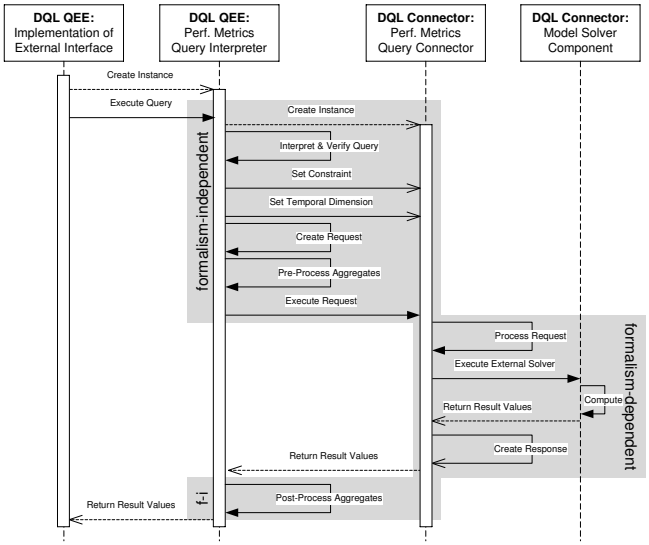

Figure 15: Execution of a Performance Metrics Query

transformed into an instance of the Mapping Meta-Model. When the necessary preparation of the request is done, the request is submitted for execution to the chosen DQL Connector. Here, steps are specific to a performance modeling formalism and prediction technique. The implementation of the DQL Connector is not constrained to specific tools but may use any kind of external service to obtain the result. When the result is available, it is represented as an instance of the Mapping Meta-Model and returned to the DQL QEE. In the third phase, post-processing operations take place. Here, the instance of the Mapping Meta-Model can be altered before it is sent back to the user as result set. This is the case if, e.g., aggregate functions are involved.

4.4 Developing DQL Connectors

To contribute a new DQL Connector, a developer has to accomplish three major tasks: (i) Create a new OSGi Bundle providing a `ConnectorProvider` as OSGi Service to the run-time, (ii) provide implementations for the relevant query classes and (iii) deploy the OSGi Bundle to the DQL environment.

For (i), the Bundle needs to be created and meta information needs to be added using the OSGi Component definition. The meta information is used to identify a DQL Connector and to register it with the DQL environment.

In (ii) the effort to implement the query classes depends on various factors and the underlying performance modeling formalism and prediction technique. `ConnectorProvider` is a factory class to create instances of classes implementing interfaces for each supported query class. For each query class a specialized interface exists, e.g. `ModelStructureQueryConnector` or `PerformanceMetricsQueryConnector`. Using this approach, in later revisions additional query classes can be added to DQL, while the compatibility with existing implementations is ensured. Within an interface, developers are free to implement only those methods for requests that should be handled by the resulting DQL Connector. For each method that requests the computation of a query, there is a corresponding method that asks for the support of the request. The DQL QEE checks for support of query requests before their invocation and in case of an unsupported method, it throws an exception. Thus, users are not required to implement each method.

To support basic queries, a DQL Connector needs to map the concepts of `Resource`, `Service` and `Probe` of the Mapping Meta-Model to representations in the performance modeling formalism that should be connected to DQL. As an entry point, we provide a DQL Connector skeleton that can be used to implement a custom DQL Connector. The skeleton contains all relevant meta-data and examples for the interaction with the DQL QEE.

Finally, after the implementation is done, in (iii) the OSGi Bundle has to be added to the run-time and queries can then be executed. As the DQL architecture is compliant to the OSGi Lifecycle Layer, a Bundle can be deployed and removed on demand.

5. RELATED WORK

DQL is related to existing work in the performance engineering domain and the model-driven engineering domain.

Intermediate Performance Models. We analyzed existing approaches in Software Performance Engineering (SPE) to reduce the efforts to transform performance models, to trigger predictions and to extract performance metrics [30]. The approaches focus on intermediate modeling techniques. The intermediate performance modeling techniques aim to generalize the transformation process from architecture-level performance models to predictive performance models utilizing an intermediate step. The approaches cope with the problem of having n input formats and m output formats (*N-to-M problem*) by providing an intermediate meta-model. The idea is that existing transformations from the intermediate meta-model to predictive performance models can be reused only by providing a transformation from an architecture-level performance model to the intermediate representation. Thus, the intermediate meta-model represents a generic performance abstraction for architecture-level performance models. Prominent examples are PMIF [32, 31], Core Scenario Model (CSM) [38] and Kernel LAnguage for PErformance and Reliability analysis (KLAPER) [10].

However, these approaches are focused on offline settings with predefined transformations, do not expose a unifying API and provide no means for a fine-granular result specification that can be used for tailored transformations as we propose with DQL.

Modeling of Degrees-of-Freedom. The analysis of DoFs in SPE is a challenging task that is common in performance prediction scenarios. In [19], DoFs are formalized for multi-objective optimization problems. Another approach is the Adaptation Points Meta-Model presented in [14]. It is mainly used to annotate model instances of system resource environments with DoFs that represent system configuration and resource allocation options. The approach also provides a language to express reconfiguration strategies [15]. Other methods for the exploration of the configuration space include, e.g., adaptive sensitivity analyses [37].

DQL is designed to integrate existing DoF modeling and corresponding exploration strategies. The mentioned approaches can thus be re-used with DQL.

Modeling Performance Metrics. As one approach for the modeling of metrics, the Objects Management Group (OMG) introduced the Structured Metrics Metamodel (SMM) as part of their Architecture Driven Modernization (ADM) roadmap [25]. Using SMM, any kind of structured metric can be modeled, measured and represented. As an example for the implementation of SMM, Measurement Architecture for Model-Based Analysis (MAMBA) [7, 6] supports a wide-range of SMM features. One notable addition of MAMBA is MAMBA Query Language (MQL) as interface to access the metrics. MQL has a Structured Query Language (SQL)-like textual syntax and supports aggregates on measurements.

However, the generic SMM and MAMBA approaches come with a significant overhead to model metric values because they are not focused on performance but aim at general analyses. They do not provide the necessary means to trigger performance predictions, but are an option for internal operations in DQL.

Domain-specific Languages. The structure of DQL is significantly influenced by the structure of SQL. SQL is a prominent example for a DSL as a query language. It allows hiding the actual data access and calculation from the user [4] and can thus be considered as role model.

6. EVALUATION

We evaluate the applicability and benefits of the proposed approach in the context of several representative case studies. We show how to query different established performance prediction techniques such as PCM [2], KLAPER [10] or QPNss, and how to query performance data repositories such as VMware vSphere. The results of our evaluation are presented in the following.

6.1 DQL Connector for PCM

Palladio Component Model (PCM) is a modeling language for component-oriented software systems and their deployment on resource landscapes. It is used to predict the performance of systems already at design time. The approach addresses the needs of users during design, development and maintenance phases of software systems and [29, 2]. PCM is ranked as a mature approach for performance modeling with sophisticated prediction techniques and comprehensive tool support [20]. PCM has been used successfully to model several different classes of component-oriented software systems [13, 28]. The Palladio Bench[4] is an Eclipse-based set of tools for modeling instances of PCM and to execute simulations of these models. With the Palladio Bench, users can develop model instances using editors with an Unified Modeling Language (UML)-based graphical syntax.

In our evaluation we demonstrate how to use PCM with DQL. The case study shows how DQL queries can be used to trigger complex performance predictions.

For the development of a DQL Connector for PCM, we analyze the meta-model of PCM and evaluate how to map PCM entities to the DQL Mapping Meta-Model. In the *Usage Model* of PCM, workload-specific modeling aspects are modeled by domain experts to represent usage scenarios of the modeled software system [29, p. 159 ff.]. We identify the types `UsageScenario` and `EntryLevelSystemCall` as relevant entities in the PCM Usage Model to be mapped to the type `Service` in the Mapping Meta-Model. In addition, PCM provides a *Repository Model* to model and store software components with their behavioral descriptions [29, p. 108 ff., p. 134 ff.]. Here, the type `ExternalCallAction` is relevant and mapped to the `Service` type of the Mapping Meta-Model. `ExternalCallAction` represents the call of a service provided by another component. Finally, in the *Resource Environment Model* of PCM the types `ProcessingResourceSpecification` and `CommunicationLinkResourceSpecification` can be mapped to the Mapping Meta-Model type `Resource`. The first type represents active resources of a hardware server, the latter type represents the network link of a hardware server.

In the DQL Connector implementation, we use the operations of the EMF API and Object Constraint Language (OCL) queries to access PCM instances and to obtain all necessary information for *Model Structure Queries*. For *Performance Metrics Queries*, we use the PCM Experiment Automation API to trigger performance predictions and the *SensorFramework*, which is part of the Palladio Bench, to obtain performance metrics. For the mapped type `Resource`, the *demanded time* and *utilization* are available as performance metrics, and for the mapped type `Service`, the metric *response time* is available. Furthermore, there are mappings for specific types of PCM to the type `DoF`.

[4] `http://www.palladio-simulator.com/tools/`

```
SELECT AppServer_CPU.utilization,
  DBServer_CPU.utilization,
  DBServer_HDD.utilization
EVALUATE DOF
  VARYING '_TyV-MFBwEd6ActLj8Gdl_A'
            AS ClosedWorkloadPopulation <100, 200>
          '_Q8jwMEg9Ed2v5eXKEbOQ9g'
            AS ActionReplication <2, 8>
FOR RESOURCE '_5uTBUBpmEdyxqpPYxT_m3w@CPU'
     AS AppServer_CPU,
    RESOURCE '_tVi40Dq_EeCCbpF63PfiyA@CPU'
     AS DBServer_CPU,
    RESOURCE '_tVi40Dq_EeCCbpF63PfiyA@HDD'
     AS DBServer_HDD
USING pcm@'mediastore.properties';
```

Listing 10: Complex DoF Query in the Palladio Bench

The DQL Connector implementation is evaluated in a case study using the MediaStore example[5] [9]. First, we analyze the model structure of the example to obtain the necessary identifiers of the model entities in the MediaStore to trigger a performance prediction. Listing 10 is a DoF Query applied to the MediaStore example. Fig. 16 shows the DQL Workbench consisting of a query editor with the DoF Query in the upper half and a view to visualize results in the lower half. The query editor provides syntax highlighting as well as auto-completion features. The visualization is a tabular representation of the Mapping Meta-Model instance returned from the DQL API call. The query contains the request to vary two DoFs with two different DoF settings each. The first DoF references the workload intensity setting for the simulation using a closed workload, the second DoF modifies the behavior of a component by replicating an internal action. The query leads to a total of four PCM simulations and four independent result sets. For each simulation run, the result set contains the utilization rates of the *AppServer_CPU*, *DBServer_CPU* and *DBServer_HDD* together with the DoF setting for the simulation. All entities in the query, i.e. `Resources` and `DoFs`, are directly referenced from the MediaStore and the performance metrics are extracted from the SensorFramework after the simulation runs complete. As a complete PCM instance consists of several sub-models, they are referenced in a properties file as the model loction in the `USING` expression.

6.2 DQL Connector for KLAPER

KLAPER aims to generalize the transformation process from architecture-level performance models to predictive performance models through an intermediate step [10]. The approach copes with the problem of having n input formats and m output formats (N-to-M problem) by providing an intermediate meta-model. The idea is that existing transformations from the intermediate meta-model to predictive performance models can be reused only by providing a transformation from an architecture-level performance model to the intermediate representation. An implementation of the KLAPER approach is available as KlaperSuite [5].

Since the KLAPER intermediate meta-model represents a generic performance abstraction for architecture-level performance models and already provides support for several

[5] `http://sdqweb.ipd.kit.edu/wiki/PCM_3.3/Example_Workspace`

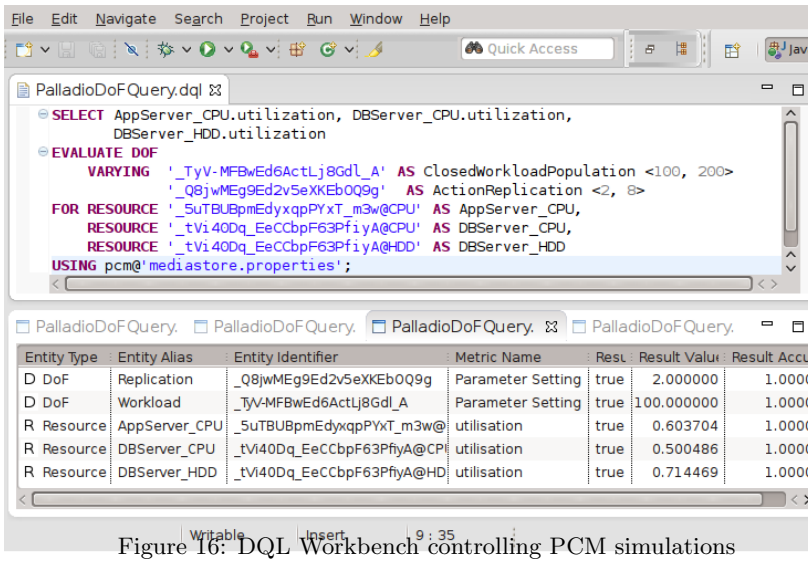

Figure 16: DQL Workbench controlling PCM simulations

modeling formalisms, it lends itself as a target model for a DQL Connector to evaluate the applicability of the DQL modeling abstractions.

We analyze the KLAPER meta-model and evaluate how to map its entities to the DQL Mapping Meta-Model. The KLAPER Meta-Model consists of different parts: (i) resources and their interaction, (ii) services and their dependencies, and (iii) the behavioral specification of services [10]. The behavior is specified in steps, which have properties to describe resource demands and performance and reliability characteristics. The Mapping Meta-Model captures only structural properties of performance models. The KLAPER types `Resource` and `Service` can be directly mapped to the Mapping Meta-Model types `Resource` and `Service`. Most of the behavioral types can be omitted in this mapping, but `ActualParam` and `Workload` can be used as `DoF` in the Mapping Meta-Model to express model variation points. Due to the nature of KLAPER as intermediate modeling language, the available performance metrics for `Resource` and `Service` depend on the used analysis model.

In summary, the mapping shows similarities of the structural abstractions used in KLAPER and the DQL Mapping Meta-Model. Furthermore, the KlaperSuite approach would benefit from an API to automate prediction processes and a mechanism to visualize results of performance predictions [5]. These points make KLAPER a valuable choice for a future DQL Connector implementation.

6.3 DQL Connector for VMware vSphere

VMware vSphere is a popular management product for VMware virtualization environments [36, 8]. vSphere offers several proprietary tools and an API that is being exposed through a web service [35]. We utilize vSphere as a performance data repository to obtain performance data from running vSphere environments.

We experienced several shortcomings of the vSphere API. The extraction of performance data needs a significant amount of repetitive source code, even for single data requests. The API and the underlying data model have a generic structure, which is hard to access in a simple way. To solve these shortcomings, we propose a DQL Connector implementation for vSphere. With vSphere, we evaluate the applicability of DQL to performance data repositories.

vSphere's underlying data center model organizes the data center resources in a deep hierarchy. The model covers the range from data center infrastructure down to resources of VMs like virtual Central Processing Units (vCPUs) and memory. In addition to live monitoring data, vSphere maintains historical performance data that can be accessed by corresponding API calls. The vSphere resource data structures can be mapped to the `Resource` type of the DQL Mapping Meta-Model. The Mapping Meta-Model does not provide means to represent the hierarchy represented in vSphere, but each vSphere resource layer can be mapped to an instance of `Resource`. There are no vSphere model elements that are mapped to the `Service` type of the Mapping Meta-Model.

With a DQL Connector for vSphere, the monitoring data access is simplified. The feature to query historical data as presented in Sec. 3.3.3 eases the access of historical data. Listing 11 is an example of an issued DQL query. The example retrieves information about the CPU and network utilization of a physical server and the mean vCPU utilization of two VMs deployed on this host. The utilization is obtained from a twelve hour time frame, in samples of one hour.

6.4 DQL Connector for Queueing Petri Nets

To show the general applicability of DQL and especially the DQL Mapping Meta-Model, we use the QPN formalism as target for an evaluation together with SimQPN as tool for simulating QPNs [1, 17]. Our goal is to show that even predictive performance models can be mapped to the Mapping Meta-Model and an integration with a DQL Connector is possible. Fig. 17 shows an example of a QPN model representing parts of the SPECjAppserver2004 benchmark application. There is an application server WLS accessing a database server DBS to serve customer requests. We refer

```
SELECT host.cpuUtilization, host.netUtilization
       MEAN(vm1.cpuUtilization,
            vm2.cpuUtilization)
FOR RESOURCE 'hostId' AS host,
    RESOURCE 'vm1Id' AS vm1,
    RESOURCE 'vm2Id' AS vm2
USING vsp@location
OBSERVE
  BETWEEN '2013-09-18 09:00:00' AND +12h
  SAMPLED BY 1h;
```

Listing 11: Example of a Basic Query for VMware vSphere

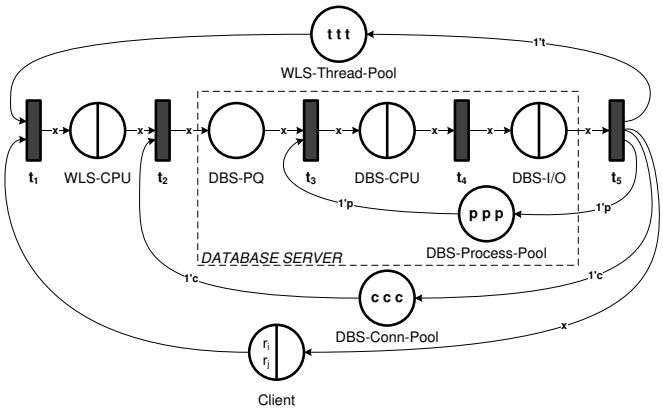

Figure 17: Part of SPECjAppserver2001 based on [17]

to [17] for the formal description of the referenced QPN and focus on the description of the mapping to DQL.

The example consists of the *queueing places* WLS-CPU, DBS-CPU and DBS-I/O. Each of these nodes consist of a queue, a scheduling strategy and a departure process. Queues are used to store *tokens* that are selected by a scheduling strategy to enter the departure process. The departure process depends on the token *color*, the token color's assigned processing time, and the number of available *servers*. We consider a queueing place in QPNs as an active resource to process requests and thus map it to the `Resource` type of the Mapping Meta-Model. The node Client is a special kind of queueing place and is used to model the clients in the system representing a *closed workload*. The processing time at this node is considered to be the *think time* required by clients to issue new requests. Client can also be mapped as active resource. The next family of model elements are *ordinary places*, i.e. WLS-Thread-Pool, DBS-PQ, DBS-Process-Pool and DB-Conn-Pool. These resources can be considered passive resources. These elements model shared resources within the network and are populated with a specified amount of tokens, i.e., the size of a thread pool. Client requests can only traverse the *transitions* t_i if the connected resources contain tokens. Otherwise the transitions do not *fire* and requests have to wait. Passive resources are mapped to the type `Resource`.

To model the workload imposed to the system, the Client queueing place is used. For a workload mix, different workload classes are specified as tokens of different colors. For each workload class $r_i, i \in \mathbb{N}$ a predefined amount of tokens is populated within the Client queue and, as different workload classes impose different processing steps, at each queueing

```
SELECT wls.utilization, order.serviceTime,
       dbs.utilization, dbsIo.utilization,
       dbsProc.population
FOR RESOURCE 'wlsCpuId' AS wls,
    RESOURCE 'dbsCpuId' AS dbs,
    RESOURCE 'dbsPoolId' AS dbsProc,
    RESOURCE 'dbsIo' as dbsIo,
    SERVICE 'newOrder' AS order
USING connector@location;
```

Listing 12: Example of a Basic Query for SimQPN

place the processing demand is parameterized accordingly. In DQL notation, $r_i, r_j, i \neq j$ represent two different instances of `Service` in the Mapping Meta-Model and i, j are considered as absolute identifiers of a `Service` instance.

After the type mapping of model entities is established, we describe how we obtain the available performance metrics. The mapping of available performance metrics is based on SimQPN [17]. For the different workload classes r_i, the *mean service time* is available. For the queueing places and ordinary places, the *token population*, *utilization*, *throughput* and *residence time* are available.

Listing 12 is an example to trigger a performance prediction for the QPN shown in Fig. 17. Here, the query is used to analyze the WLS-CPU and the service time of requests from the workload class *newOrder*. To determine if the *Database Server* is a bottleneck in this model, the utilization rates of DBS-CPU and DBS-IO are requested. Additionally, the population of DBS-Process-Pool is requested. In this scenario, either the resources of the Database Server are saturated or it is the process pool that is too small and thus a bottleneck. Furthermore, using DoFs the population of client can be varied to capture the behavior for different load levels.

7. CONCLUSIONS & FUTURE WORK

We presented the Descartes Query Language (DQL), a novel query language to specify performance queries. It unifies the interfaces of available performance modeling formalisms and their prediction techniques to provide a common Application Programming Interface (API). DQL is independent of the employed modeling formalisms, hides low-level details of performance prediction techniques and thus reduces the manual effort and the learning curve when working with performance models. DQL allows expressing simple performance queries, that specify which performance metrics should be predicted under which scenario-specific constraints, and also allows complex queries that trigger an automated exploration of the configuration space of a performance model for a sensitivity analysis.

We implemented our approach using an extensible architecture. Support for specific performance modeling formalisms and prediction techniques can be easily integrated by adding new *DQL Connectors*. To demonstrate the applicability and benefits of our approach, we presented representative case studies for DQL showing how to query different established performance prediction techniques such as PCM [2], KLAPER [10] or Queueing Petri Nets (QPNs), and how to query performance data repositories such as VMware vSphere.

As part of our future work, we plan to integrate further DQL Connectors and to provide additional query classes. Additional query classes in DQL are intended to address problems like the automated detection of bottlenecks. These classes are intended as a step towards *goal-oriented queries* that can be used to express optimization problems. Furthermore, we encourage researchers and tool providers to contribute their own DQL Connectors to ease the usage of their prediction techniques.

8. REFERENCES

[1] F. Bause. Queueing Petri Nets - A formalism for the combined qualitative and quantitative analysis of systems. 1993.

[2] S. Becker, H. Koziolek, and R. Reussner. The Palladio component model for model-driven performance prediction. *Journal of Systems and Software*, 2009.

[3] F. Brosig, N. Huber, and S. Kounev. The Descartes Meta-Model. Technical report, Karlsruhe Institute of Technology (KIT), Karlsruhe, 2012.

[4] D. D. Chamberlin and R. F. Boyce. SEQUEL. In *FIDET '76*, New York, New York, USA, 1976. ACM Press.

[5] A. Ciancone, M. L. Drago, A. Filieri, V. Grassi, H. Koziolek, and R. Mirandola. The KlaperSuite framework for model-driven reliability analysis of component-based systems. *Software & Systems Modeling*, 2013.

[6] S. Frey, R. Jung, B. Kiel, and W. Hasselbring. MAMBA: Model-Based Software Analysis Utilizing OMG's SMM. 2012.

[7] S. Frey, A. van Hoorn, R. Jung, W. Hasselbring, and B. Kiel. MAMBA: A measurement architecture for model-based analysis. 2011.

[8] Gartner Inc. Magic Quadrant for x86 Server Virtualization Infrastructure, 2013.

[9] F. Gorsler. Online Performance Queries for Architecture-Level Performance Models. Master's thesis, Karlsruhe Institute of Technology (KIT), 2013.

[10] V. Grassi, R. Mirandola, E. Randazzo, and A. Sabetta. KLAPER: An Intermediate Language for Model-Driven Predictive Analysis of Performance and Reliability. Lecture Notes in Computer Science. Springer Berlin / Heidelberg, 2008.

[11] J. Happe, H. Groenda, M. Hauck, and R. H. Reussner. A Prediction Model for Software Performance in Symmetric Multiprocessing Environments. QEST '10, pages 59–68. IEEE Computer Society, 2010.

[12] N. R. Herbst, N. Huber, S. Kounev, and E. Amrehn. Self-Adaptive Workload Classification and Forecasting for Proactive Resource Provisioning. In *ICPE 2013*, New York, NY, USA, April 2013. ACM.

[13] N. Huber, S. Becker, C. Rathfelder, J. Schweflinghaus, and R. H. Reussner. Performance modeling in industry. In *ICSE '10*, New York, New York, USA, 2010. ACM Press.

[14] N. Huber, F. Brosig, and S. Kounev. Modeling dynamic virtualized resource landscapes. In *QoSA '12*, New York, New York, USA, 2012. ACM Press.

[15] N. Huber, A. van Hoorn, A. Koziolek, F. Brosig, and S. Kounev. Modeling Run-Time Adaptation at the System Architecture Level in Dynamic Service-Oriented Environments. *Service Oriented Computing and Applications Journal (SOCA)*, 2013.

[16] S. Kounev, F. Brosig, N. Huber, and R. Reussner. Towards Self-Aware Performance and Resource Management in Modern Service-Oriented Systems. In *SCC '10*. IEEE, 2010.

[17] S. Kounev and A. Buchmann. SimQPN - A tool and methodology for analyzing queueing Petri net models by means of simulation. *Performance Evaluation*, 2006.

[18] A. Koziolek, D. Ardagna, and R. Mirandola. Hybrid multi-attribute QoS optimization in component based software systems. *Journal of Systems and Software*, 86(10), 2013.

[19] A. Koziolek and R. Reussner. Towards a generic quality optimisation framework for component-based system models. *CBSE '11*, 2011.

[20] H. Koziolek. Performance evaluation of component-based software systems: A survey. *Performance Evaluation*, 2010.

[21] J. Li, J. Chinneck, M. Woodside, M. Litoiu, and G. Iszlai. Performance model driven QoS guarantees and optimization in clouds. In *ICSE '09*. IEEE, 2009.

[22] D. A. Menasce, L. W. Dowdy, and V. A. F. Almeida. *Performance by Design: Computer Capacity Planning By Example*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 2004.

[23] Object Management Group. UML Profile for Schedulability, Performance, and Time Specification 1.1. 2005.

[24] Object Management Group. Meta Object Facility (MOF) 2.0 Query/View/Transformation, 2011.

[25] Object Management Group. Structured Metrics Metamodel 1.0, 2012.

[26] OSGi Alliance. OSGi Service Platform Core Specification - Release 4, Version 4.3. 2011.

[27] OSGi Alliance. OSGi Service Platform Service Compendium - Release 4, Version 4.3. 2012.

[28] C. Rathfelder, S. Becker, K. Krogmann, and R. Reussner. Workload-aware System Monitoring Using Performance Predictions Applied to a Large-scale E-Mail System. In *WICSA/ECSA '12*. IEEE, 2012.

[29] R. Reussner, S. Becker, E. Burger, J. Happe, M. Hauck, A. Koziolek, H. Koziolek, K. Krogmann, and M. Kuperberg. The Palladio Component Model. Technical report, Karlsruhe Institute of Technology (KIT), Karlsruhe, 2011.

[30] C. Smith and C. Lladó. Model Interoperability for Performance Engineering: Survey of Milestones and Evolution. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2011.

[31] C. U. Smith, C. M. Llado, and R. Puigjaner. Model Interchange Format Specifications for Experiments, Output and Results. *The Computer Journal*, 2010.

[32] C. U. Smith, C. M. Lladó, and R. Puigjaner. Performance Model Interchange Format (PMIF 2): A comprehensive approach to Queueing Network Model interoperability. *Performance Evaluation*, 2010.

[33] C. U. Smith and L. G. Williams. *Performance Solutions - A Practical Guide to Creating Responsive, Scalable Software*. Addison-Wesley, 2002.

[34] E. Thereska, D. Narayanan, and G. Ganger. Towards Self-Predicting Systems: What If You Could Ask "What-If"? In *DEXA '05*. IEEE, 2005.

[35] VMware Inc. VMware vSphere Web Services SDK, 2012.

[36] VMware Inc. vSphere Monitoring and Performance, 2013.

[37] D. Westermann, J. Happe, R. Krebs, and R. Farahbod. Automated inference of goal-oriented performance prediction functions. In *ASE 2012*, ASE 2012, New York, New York, USA, 2012. ACM Press.

[38] M. Woodside, D. C. Petriu, D. B. Petriu, H. Shen, T. Israr, and J. Merseguer. Performance by unified model analysis (PUMA). In *WOSP '05*, New York, New York, USA, 2005. ACM Press.