Karlsruhe Institute of Technology

Online Performance Queries for Architecture-Level Performance Models

Master Thesis of

Fabian Gorsler

At the Department of Informatics
Institute for Program Structures
and Data Organization (IPD)

Reviewer:	Prof. Dr. Ralf H. Reussner
Second reviewer:	Prof. Dr. Walter F. Tichy
Advisor:	Dipl.-Inform. Fabian Brosig
Second advisor:	Dr.-Ing. Samuel Kounev

Duration: 2012-12-12 – 2013-07-09

I declare that I have developed and written the enclosed thesis completely by myself, and have not used sources or means without declaration in the text.

Karlsruhe, 2013-07-09

.....

(**Fabian Gorsler**)

Abstract

As modern software systems are subject to increasing dynamics, achieving an acceptable performance becomes a challenge of increasing concern. Thus, during design- and run-time, the performance of a software system needs to be analyzed continuously to avoid contention. Model-based performance prediction is an approach to analyze software systems by predicting their behavior and supporting users to draw conclusions.

Available approaches for performance prediction are usually based on their own modeling formalism and analysis tools. Users are forced to gain detailed knowledge about these approaches before predictions can be made. To lower these efforts, intermediate modeling approaches simplify the preparation and triggering of performance predictions. However, users still have to work with different tools suffering from integration, providing non-unified interfaces and the lack of interfaces to trigger performance predictions automatically.

Our approach is to provide a novel query language capable of expressing queries for questions like "What is the response time of service X?". Previous shortcomings are addressed by an interface to integrate different tools. The interface is accessible through a unified query language to trigger performance predictions. The design of the query language is based on a classification scheme with an implementation of an extensible architecture aiming to integrate a broad range of tools and third-party extensions.

The query language is evaluated by integrating a prominent approach for performance prediction and controls it while conducting a performance analysis. Two additional approaches are conceptually evaluated for integration and showing promising synergies. Overall the results are encouraging and motivate the future development of our query language and the integration of additional tools.

Zusammenfassung
Da moderne Software-Systeme steigender Dynamik unterworfen sind, ist es eine immer bedeutsamere Herausforderung ein akzeptables Leistungsverhalten zu erzielen. Aus diesem Grund ist es, während der Entwurfszeit und im Betrieb, notwendig das Leistungsverhalten kontinuierlich zu analysieren um Engpässe zu verhindern. Modellbasierte Leistungsvorhersage ist ein Ansatz, Software-Systeme durch Vorhersagen zu analysieren und Benutzer beim Rückschließen zu unterstützen.

Verfügbare Ansätze zur Leistungsvorhersage basieren für gewöhnlich auf eigenen Modellierungsverfahren und Analysewerkzeugen. Benutzer werden daher dazu gezwungen Detailkenntnisse über diese Ansätze zu entwickeln bevor sie Vorhersagen erstellen können. Um diesen Aufwand zu reduzieren, sowie die Vorbereitung und das Starten von Vorhersagen zu vereinfachen, wurden Ansätze auf Basis von Zwischenmodellen entwickelt. Nichtsdestotrotz bleiben die Probleme, dass Benutzer mit verschiedenen, nicht integrierten Werkzeugen arbeiten müssen, uneinheitliche Schnittstellen bereitgestellt werden und Schnittstellen für eine Automatisierung von Leistungsvorhersagen fehlen.

Unser Ansatz ist es eine neuartige Abfragesprache zu entwickeln. Die Abfragesprache erlaubt es eine Frage wie „Was ist die Antwortzeit von Dienst X?," auszudrücken. Einschränkungen aus früheren Arbeiten werden durch eine integrale Schnittstelle gelöst, die verschiedene Werkzeuge bündelt und es durch eine einheitliche Abfragesprache erlaubt Leistungsvorhersagen auszuführen. Der Entwurf der Abfragesprache basiert auf einem Klassifikationsschema, sowie einer Implementierung auf Basis einer erweiterbaren Architektur um ein breites Band von Verfahren und Erweiterungen einzubinden.

Zur Evaluierung der Abfragesprache integrieren wir einen bekannten Ansatz für die Leistungsvorhersage und führen eine Leistungsanalyse durch die Abfragesprache aus. Zwei weitere Ansätze werden konzeptionell für eine spätere Integration mit der Abfragesprache betrachtet und zeichnen ein erfolgversprechendes Bild ab. Insgesamt sind die bisherigen Ergebnisse unseres Ansatzes zufriedenstellend und motivieren die zukünftige Weiterentwicklung der Abfragesprache sowie die Integration weiterer Werkzeuge.

Contents

List of Figures

List of Tables

Listings

1. Introduction

Modern Information Technology (IT) environments and systems are expected to provide tailored services to customers in a responsive manner. Performance and availability are of fundamental importance for today's software systems [MDA04, SW02]. On the other side, modern IT solutions are introducing additional abstraction layers leading to an increasing complexity of these systems, which makes achieving an adequate performance challenging.

To avoid performance problems within a software system, it is important to analyze its performance characteristics during all phases of its life-cycle. Software Performance Engineering (SPE) is a discipline that focuses on *"systematic, quantitative approaches to the cost-effective development of software systems to meet performance requirements"* [SW02]. At each phase of the life-cycle of a software system, SPE helps to estimate the level of performance a system can achieve and helps to make recommendations to improve the performance level [MDA04].

While during the design the choice of components and algorithms is of interest and their behavior needs to be analyzed, at operation the dynamic allocation of processing resources to handle customer workloads becomes more important [BHK12]. Achieving not a sufficient level of performance during operation, providers might suffer from not being able to process all incoming customer requests and to miss business goals. For both cases predictions are needed to make an assumption on the behavior of the software system and to be able to act before the system is saturated.

Typically, in SPE performance models are used to predict the performance of software systems [Kou05]. A performance model is an abstract representation of a software system and may also include the resources used by a software system. The advantage of the use of performance models is, that they allow to make predictions before a software system is finished and that the cost is lower than providing a real system for experimentation [MAD94].

1.1. Motivation

Several approaches for model-based performance prediction exist [BDIS04, Koz10]. The approaches differ in their modeling formalism, their expressiveness and their model solving techniques. The available approaches can be categorized as predictive performance models and descriptive performance models with architecture-level information.

Predictive performance models capture the temporal system behavior and can be used for performance prediction by means of analytical or simulation techniques. To simulate the

behavior of software systems using predictive performance models various approaches exist. One example of a family of modeling formalism are queuing networks and their specializations [MDA04, Kou06, LCW+09]. Descriptive, architecture-level performance models describe performance-relevant aspects of software system at the software architecture level. These can normally be transformed automatically into predictive performance models allowing to predict the system performance for a given workload and configuration scenario. Prominent examples are the UML SPT profile [Obj05] and its successor the UML MARTE profile [Obj11b], the CSM [WPP+05], PCM [BKR09] and KLAPER [GMRS08].

Available approaches for model-based performance predictions usually provide their own toolchain and proprietary interfaces to trigger performance predictions, e.g. [SKM12, CDF+13]. Thus, performance analysts are demanded to learn the modeling formalism of the used approach and have to get familiar with the available toolchain for triggering performance analyses. A common interface to access common performance prediction tools is desirable.

The efforts for a performance prediction are high, which motivates the development of approaches to unify performance predictions by using intermediate modeling steps [WPP+05, GMRS08]. These approaches aim to reduce the effort of transforming descriptive performance models to predictive performance models by addressing the N-to-M problem of n descriptive performance models to be analyzed by m suitable predictive performance models. Instead of transforming an architecture-level model directly to a predictive model, the authors propose an intermediate model as transformation target. Using these approaches, existing transformations from the intermediate model to predictive performance models can be reused. However, the approaches does not provide a generic interface to trigger performance predictions. Performance analysts still need to deal with multiple solving and analysis tools, and in case of automated performance analysis, toolchains suffer from missing Application Programming Interfaces (APIs) to trigger predictions programmatically.

Furthermore in online prediction scenarios as in [KBHR10], performance analysts are often faced with questions like:

- What performance would a newly deployed service exhibit and how much resources should be allocated to it?
- What would be the effect of migrating a service from one Virtual Machine (VM) to another?
- How should the system configuration be adapted to avoid performance problems arising from increasing customer workloads?

We refer to these questions as *online performance queries* and address the previously mentioned shortcomings by providing a novel interface to trigger performance predictions. Our approach is based on a query language as an interfaces that (i) allows the description of performance queries, (ii) eases manual usage, and (iii) provides an API to trigger queries in a programmatic way.

1.2. Aim of the Thesis

The aim of the thesis is the development of a language for performance queries. We focus on online prediction scenarios with architecture-level performance models. However, this allows also the usage for predictions at system development and deployment time. We pursue the following goals:

- Provide an overview of common usage scenarios for performance predictions.

- Identify different classes of performance queries and provide a corresponding classification scheme.
- Propose expressions to form a query language for the analyzed usage scenarios.
- Design and implement a toolchain to connect the proposed query language as unifying interface to existing performance prediction approaches.
- Implement the query language with pre- and post-processing steps allowing orthogonal features, expose extension points for third-party contributions and provide users a graphical interface to execute queries.
- Evaluate the design and implementation with established performance modeling approaches to demonstrate the applicability and appropriateness of the query language.

The query language includes features to address concerns during the design-time of software systems, i.e. to configure a sensitivity analysis of variable parameters, and during the run-time, i.e. to configure a trade-off of prediction timeliness vs. accuracy. It captures the demands of manually querying a performance model, i.e. to browse a performance model and to build queries incrementally, and to automate performance predictions, i.e. to prepare or to generate queries by applications.

For the integration of existing interfaces we provide a toolchain consisting of an editor to be used for building queries, exemplary implementations for architecture-level performance models and interfaces to process queries from arbitrary sources. The implementation is realized with a special focus on an extensible architecture to integrate available approaches for model-based performance predictions in subsequent development steps.

We evaluate the applicability of our query language in several steps. First the design of the query language is evaluated by providing a showcase of a workflow for performance analysis, exemplary queries and descriptions of the expressiveness of queries. Following, the query language is evaluated by providing an integration of an established example of an architecture-level performance model [BKR09, Koz10]. Furthermore, we show how the query language can be used to employ intermediate performance models as they are presented in [GMRS08], and how the queries can be integrated into an existing experimentation framework for sensitivity analysis[WHHH10].

1.3. Document Outline

In Chapter 2 an introduction of relevant technologies is given. The introduced technologies build a foundation for the design of the Descartes Query Language (DQL) and its implementation.

Chapter 3 focuses on the design of DQL. We describe a set of motivating usage scenarios and user stories. The user stories will be leveraged to derive a classification scheme for queries and to classify the resulting functionality of queries. The designated query classes are used to formulate a conceptual design for the implementation. The chapter is completed by the detailed description of the resulting grammar used to form DQL.

Chapter 4 provides an overview of the state of the art of related work in research. An additional excerpt introducing Structured Query Language (SQL) as Domain-specific Language (DSL) concludes the chapter.

Chapter 5 is focused on the implementation of DQL. The toolchain used for the implementation is based on Xtext as a framework for creating DSLs, Eclipse Modeling Framework (EMF) for the modeling of implementation parts and OSGi as run-time environment offering the foundation to realize an extensible software architecture. Besides of the static

software architecture, platform dynamics and the execution of queries are explained in detail.

To demonstrate the capabilities and the appropriateness of DQL, an evaluation is following in Chapter 6. In the evaluation, we apply DQL to established modeling approaches like Palladio Component Model (PCM), and demonstrate further use cases of implementation artifacts to provide access to performance data repositories.

The concluding Chapter 7 summarizes the work and contribution of this thesis and gives an outlook on future work.

2. Technological Foundations

This chapter introduces technological foundations fundamental for this thesis. The first two sections focus on the field of Software Performance Engineering (SPE). In Section 2.1 an introduction to performance modeling and analyses will be given, which is completed by an introduction of the Descartes Meta-Model (DMM) and its building blocks in Section 2.2. The DMM is an example of a descriptive performance model capturing architecture-level information. The final two sections focus on technologies used in the implementation parts. Section 2.3 presents technologies used in Model-driven Software Development (MDSD) as integral parts of the implementation artifacts. The OSGi Framework as an approach for implementing component-oriented architectures will follow in Section 2.4 and complete this chapter.

2.1. Performance Modeling and Analyses

Modeling software systems can be seen as a compromise between effort for modeling, accuracy of calculated results and the amount of computing time needed. Especially when using online performance analyses to predict the behavior of a software system, the required computing time becomes an influencing factor. In addition, the selected modeling technique determines the available analysis techniques, as models have varying expressiveness.

The *Operational Analysis* is a set of common methods to evaluate the behavior of a system analytically. Common methods are, e.g. the *Service Demand Law* for determining the service time of requests at resources, the *Utilization Law* to calculate the average fraction of utilization of resources or *Little's Law* for calculating the average number of requests waiting to be processed at a specific resource. Detailed descriptions of methods of the Operational Analysis can be found in [MDA04, p. 61 ff.].

Opposed to analytic methods, simulation models can be used to predict the behavior of a system and allow to capture dynamics. The Operational Analysis is limited to static relationships of model entities. Simulation models might therefore be seen as being more expressive at the price of additional effort for modeling and solving. Furthermore the higher Degree of Freedom (DoF) through modeling comes at the danger *"...that simulation models that are not properly validated can produce useless and misleading results"* [MDA04, p. 38].

Additional model validation steps to analyze whether the desired information demand can be answered, are an essential and recurring task during the simulation model creation. In [MDA04, p. 36 ff.] a discussion of analytic methods and simulation models is given.

A common approach for simulation models and performance prediction is the use of *Queueing Network (QN) Models*, that are introduced informally in the following. A QN consists of one or more queues, connections between queues and tokens that characterize requests. Requests are processed at queues by servers and each queue has at least one server associated for completing requests at a given service time.

Depending on the actual modeling approach, the requests can be modeled as open, closed or mixed workloads to describe how requests enter and leave the system. In addition, a QN can be enhanced to support different classes of workloads, that might use different queues depending of their class. These models allow to reflect more complex systems. A formal description of QNs with additional examples can be found in [MDA04].

To compute performance metrics using a QN, several approaches exist. One approach might be to use Operational Analysis as proposed by [DB78]. Although being not as expressive as a simulation, bounds for the system can be calculated. An approach for the simulation of an extended form of QNs, SimQPN, can be found in [KB06, KSM11]. The simulation of request processing is based on stochastic distributions for service times and step-wise transitions of requests between queues. Furthermore the scheduling of requests at queues can be defined to meet the circumstances of the real-world system.

For providers of architecture-level performance models, e.g. like the DMM, one fundamental challenge is to provide transformation approaches towards analytic methods or simulation models. Tailored transformations are crucial to offer users views on their software systems and to predict performance metrics for these views. A view might be composed only of a set of model entities, as the performance analyst or software engineer is focused on a special part of the software system.

As software systems might change over time and the modeled resource allocation might be altered due to new requirements, automated transformations and analyses are needed to reason the effort for simulation purposes. Through generalized transformation rules, the effort for validation of simulation models can be reduced and might only occur during development time. Past work shows this approach is viable [WHSB01, ILFW05, BKK09].

2.2. The Descartes Meta-Model

The DMM is a novel approach for modeling modern distributed computing facilities and enterprise software systems. The main objective of the DMM is the management of Service Level Agreements (SLAs)[1] while ensuring an efficient resource allocation. Especially in modern computing environments, e.g. Cloud Computing environments, where a resource allocation on demand is possible, customers are likely to optimize their resource allocation [BHK12].

Using a dynamic resource allocation, providers can adapt their computing power to their customers' demand and therefore reduce the operating cost of their computing systems. For customers the dynamic resource allocation should be a transparent process as in an optimal scenario the response times should not change during changes in the allocation. The key challenge in this case is the self-awareness of systems to reconfigure their resource allocation. Human interaction should be reduced to a minimum effort.

The concept of the DMM is based on an architecture-level approach covering more aspects than a plain stochastic performance model. An architecture-level model covers a broader view of an environment and is able to evolve and to reflect dynamic system changes. The additional benefit of architecture-level performance models comes at the cost of higher effort for creating the model instance and maintaining it.

[1] This thesis deals only with performance relevant aspects of the DMM, other aspects are out of scope.

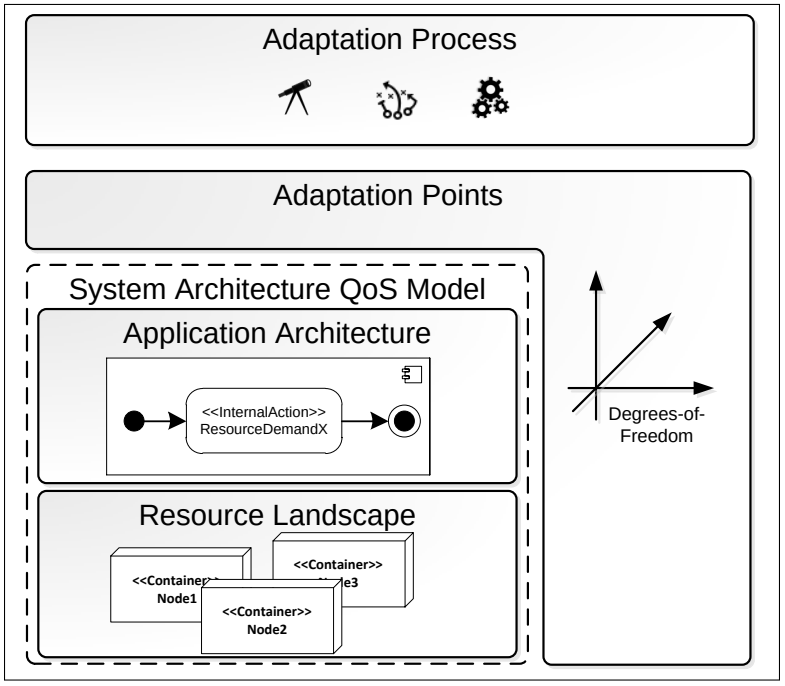


Figure 2.1.: Descartes Meta-Model Overview based on [BHK12]

An architecture-level performance model is desirable as a foundation of the Descartes Query Language (DQL) in order to provide online prediction capabilities and to reflect dynamic configuration changes of an operating environment. Furthermore having a more general approach than pure stochastic performance models as foundation, allows to cope with complex environments found in modern data centers.

From a high-level point of view the DMM should be seen as the integration of different parts as shown in figure 2.1: (i) A *Resource Landscape Sub Meta-Model* for the modeling of environments, (ii) an *Application Architecture Sub Meta-Model* to cover the performance characteristics of an application, (iii) an *Adaptation Point Sub Meta-Model* to reflect the DoF in a system environment and finally (iv) an *Adaptation Process Sub Meta-Model* to represent the logic for adaption processes.

Through using (i) and (ii) already System Architecture Quality of Service (QoS) models can be created, e.g. for evaluating capacity planning scenarios. For more sophisticated models using (iii) and (iv) complex and dynamic structures can be modeled. This way, the gap between models of software systems and models of computing systems, is bridged allowing more advanced techniques.

Examples for the modeling of DoFs in the Adaptation Point Model are options to define boundaries on the configuration space (e.g. Virtual Machine (VM) need at least one virtual Central Processing Unit (vCPU) provisioned and can operate up to eight vCPUs) and furthermore offers validation capabilities (e.g. 1 - 64 GBs of memory can be allocated, but configurations are only valid in steps of the power of 2 Bytes). Through modeling Adaptation Points of an environment, technical aspects can be formalized and evaluated as a scheme of rules.

The implementation of DMM is based on Eclipse Modeling Framework (EMF), which is introduced in Section 2.3. The meta-model part is still under ongoing development and will be extended in future. Furthermore the model is also constrained by rules using the Object Constraint Language (OCL) that brings semantics to the model and ensures valid model instances. [BHK12, Obj12]

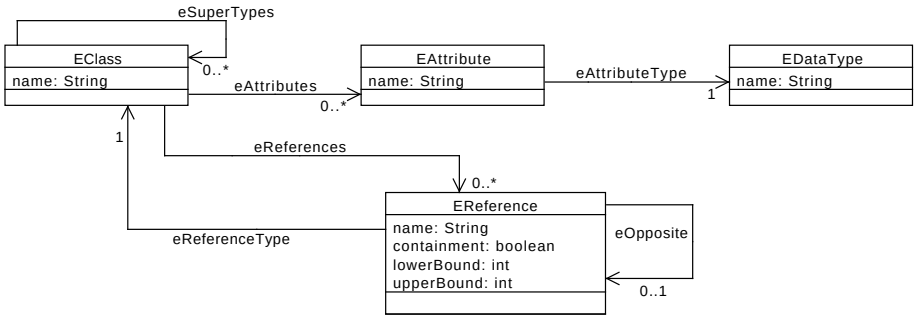


Figure 2.2.: Ecore Kernel based on [SBPM09, p. 105]

2.3. Eclipse Modeling Framework and Xtext

EMF is part of the Eclipse Modeling Project[2] providing a meta-model, called Ecore, and a toolchain for MDSD tightly integrated into the Eclipse Platform. EMF is intended to bridge pure modeling techniques like described by Model-driven Architecture (MDA) and more pragmatic approaches like MDSD [SBPM09]. It provides facilities to specify parts of applications using model-based techniques to assist engineers during the initial development and later maintenance phases. Using clean and well-defined modeling techniques, the software design is expected to benefit from precise actions that can be performed to implement changes.

The Ecore Meta-Model is made up on several parts, where the Kernel is the main part as displayed in figure 2.2. The Kernel consists of the entities required to model object-oriented classes with attributes, data types and references between classes. Further extensions of the Kernel are based up on structural and behavioral features, classifiers, packages and factories and annotations. Seen from an architectural viewpoint, Ecore features a class-concept that addresses known features of the Java programming language and makes it therefore a suitable meta-model for Java-based implementations [SBPM09, p. 106 ff.].

The foundation of EMF is derived from the principles of Unified Modeling Language (UML) and Meta Object Facility (MOF) created by the Objects Management Group (OMG). OMG defines these standards for interoperability and collaboration in software architecture, engineering and development domains. EMF and its meta-model Ecore are built up on a subset of these specifications crafted to object-oriented software development, which can be seen as Ecore is based on a subset of UML Class Diagrams [Obj11a, Obj11c, Obj11d].

For the implementation of programming languages and Domain-specific Languages (DSLs), Xtext[3] provides an integrated toolchain approach [Ecl12]. The Xtext toolchain is based on EMF and uses model-driven techniques to provide a comprehensive set of infrastructure to start the customization of a DSL environment. To provide the infrastructure, Xtext uses a grammar file in an Extended Backus-Naur Form (EBNF)-like syntax as input, which is processed to provide a tailored parser and lexer, an editor integrated with the Eclipse Integrated Development Environment (IDE) and source code skeletons to customize the environment. The language infrastructure is provided for further use as set of OSGi Bundles (see Section 2.4).

The internal representation of the parser and lexer results is stored as a EMF model instance. Using this representation of DSL statements, the language developer can interpret the model instance and implement the execution of statements. The User Interface

[2] `http://www.eclipse.org/modeling/`
[3] `http://www.eclipse.org/Xtext/`

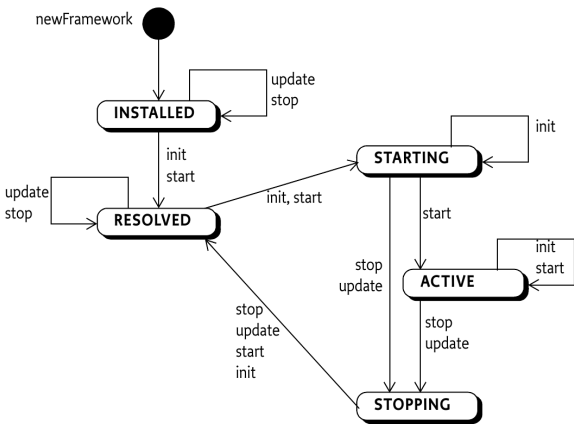


Figure 2.3.: OSGi Life Cycle Layer State Machine from [OSG11, p. 92]

(UI) parts generated through Xtext provide an editor with support for the textual syntax of a DSL implemented. The editor is capable of syntax highlighting and to provide a context-dependent assistance to complete statements. A developer may extend the context-dependent assistance with more sophisticated completion advice through implementing hooks for grammar rules.

2.4. OSGi Framework Approach

The OSGi Framework is an approach for developing, deploying and operating component-oriented developed software. Software components are contained in *OSGi Bundles*, that can be deployed to OSGi-compliant run-times. The OSGi Framework is the foundation for the development of an OSGi-compliant run-time. Examples for OSGi-compliant run times are Eclipse Equinox[4], Apache Felix[5] or IBM CICS[6]. To allow dynamic and scalable application management, each OSGi Bundle defines its depending packages and which packages it provides to the OSGi run-time. OSGi is focused on Java-based environments, therefore software packages are corresponding to Java packages. To separate concerns of the framework, it is split into several layers. A brief introduction to the *Life Cycle Layer* and the *Services Layer* will follow [OSG11, p. 7 ff.].

To support on-demand usage of software components in a run-time, OSGi specifies a *Life Cycle Layer* for OSGi Bundles. Bundles are started when functionality is required from dependent OSGi Bundles and are stopped if they are no longer in use. The life cycle management allows OSGi-based software products to scale their static overhead by present demands[OSG11, p. 79 ff.].

The life cycle management of OSGi Bundles consists of several different states. After initializing the OSGi Framework, OSGi Bundles found through the run-time are registered as *installed* OSGi Bundles. When a bundle is *starting*, its dependencies are resolved and additional resources are initialized, e.g. a Classloader for the OSGi Bundle. After a successful start the becomes *active* and is operational.

When the OSGi Bundle is *stopping*, resources are deallocated and dependent OSGi Bundles need to be stopped (i.e. if stopped forcefully). After this process completes, the OSGi Bundle stays *resolved* and can be started again. While staying resolved, the OSGi run-time is able to choose which OSGi Bundle resources to be kept alive, e.g. the Classloader

[4] `http://www.eclipse.org/equinox/`
[5] `http://felix.apache.org/`
[6] `http://www.ibm.com/software/htp/cics/`

instance, in order to save resources. The complete state machine of the OSGi Life Cycle Layer can be seen in Figure 2.3.

Using the Service Layer in OSGi applications, functionality from different OSGi Bundles can be used in a dynamic and decoupled fashion. A bundle may register services, that can be referenced from other OSGi Bundles on demand. The registration of services is based on exposing a Java interface provided by a OSGi Bundle as offered service [OSG11, p. 117 ff.]. Exposing services from OSGi Bundles and requesting services from other OSGi Bundles can be seen as an implementation approach of a Component-based Software Engineering (CBSE)-based architecture.

As the Service Layer depends on the concepts of the Life Cycle Layer, services are aware of the life cycle management and offer management operations to promote life cycle changes to services based on an event-oriented mechanism. Using a *Service Reference*, OSGi Bundles are able to request functionality of other OSGi Bundles on demand.

While a Service Reference is stored, the exposing OSGi Bundle is not needed to be active. A OSGi Bundle might stay resolved until the Service Reference is used to obtain an instance of the service. If the service is requested, the OSGi Bundle is activated. The life cycle management allows therefore an efficient dependency handling among OSGi Bundles and the OSGi run-time.

3. Design of a Query Language for Online Performance Queries

This chapter introduces the conceptual foundation and the design of DQL as a textual syntax to express online performance queries. The chapter uses a funnel-like structure starting at the definition of high-level requirements down to abstract grammar rules to specify the language. In Section 3.1 at first the prospect usage scenarios for DQL is outlined and illustrated by user stories. Based on these requirements, in Section 3.2 the functionality of queries is analyzed to form a classification scheme. Using this classification scheme, in Section 3.3 expressions will be formed as an initial textual representation, which is evaluated by exemplary queries in Section 3.4 to judge on the language structure in contrast to the initial user stories. Section 3.5 completes this chapter with detailed descriptions of the grammar rules.

3.1. Usage Scenarios and User Stories

3.1.1. Definition of Usage Scenarios and User Roles

Foundation for the design of the query language is the analysis of requirements presented in this section. As no directly comparable query language in the domain of SPE exists, this section will introduce usage scenarios and corresponding roles acting as language users. For each role, a corresponding scenario introduction and environment description is given. To derive requirements, user stories will be used. Each user story contains an objective the role wants to achieve through using the query language.

The user stories presented in this section are intended to illustrate requirements and are based on a compromise between formal requirement definitions describing each possible functionality and informal approaches for agile development practices as in [Coh10, p. 235 ff.]. The user stories and their implied functionality will be discussed for each user role.

User stories are built up on single sentences written in active language and contain a single objective, but not an exact specification on the way to achieve an objective. If user stories are too long, they are split up in multiple stories and if additional information is required to reason on user stories or to understand the scenario, the discussion will provide additional details.

In Table 3.1 all user roles, scenarios and domains for the following user stories are shown. On the first layer, user types are differentiated by human and machine users. The language

User Type	Scenario	User Role	Description
Human	offline	Software Architect	Evaluate different software architecture approaches for performance goals
		Software Engineer	Evaluate different implementation approaches, frameworks or libraries with derived models from experiment runs
	online	Performance Analyst	Investigate for optimization potential, advice resource management, coordinate SLA restoration process manually
Machine	offline	Performance Testing	Automated performance regression testing after redesign and refactorings based on automated performance model extraction and evaluation
	online	Application Server	Application-specific fine tuning of low-level configuration parameters based on actual workload demands, e.g. for thread pool sized or garbage collection
		Monitoring System	Periodic performance queries with run-time aspects and forecasting functionality for alerting and reporting
		Resource Management	Resource management helps to optimize the resource allocation for software systems during runtime and tries to find the best trade-off between response times/throughput and allocation cost

Table 3.1.: User Types and Usage Environments

shall integrate requirements for both types in order to provide an unified interface. Next, for both user types, the usage scenario might be either offline, e.g. for analyses during the design time of a software system, or online, e.g. to determine performance bottlenecks during operation in production. To support the discussions, another layer separates the user roles, e.g. into roles of software engineers, application servers or resource management systems.

The following user types, usage scenarios and roles are defined and influenced up on information from [BHK12, RBB+11, TNG05]. Further refinements of the query language design and a classification scheme will follow in the subsequent sections.

3.1.2. User Stories for Offline Scenarios

3.1.2.1. Software Architect and Software Engineer

Scenario

During design and development time several decisions have significant impact on the performance of the resulting software system. During design time the choice of components, their communication patterns and the underlying infrastructure have direct influence on performance metrics during run time. Whereas during development time the selection of algorithms and the settings of tuning and system parameters influence the performance behavior.

Environment

For both roles, the environment is an offline environment. Thus the software system and its resource allocations can be changed without any risk of SLA violations. Recurring experiments through the performance model instance and changing the instance is possible. On the other hand the workload patterns found in real workloads are not reproducible, which limits the validity of experiments. As during design time only parts of the software

system might be available, resource demands and parameterization in the performance model instance will be based on assumptions. During development time these assumptions can be refined through measurements, but the deployment for later operation might change the system configuration.

User Stories

- ⋆ **US 1:** As a user, I want the system for query execution to load a performance model instance of a specific performance meta-model.
- ⋆ **US 2:** As a user, I want explore all queryable entities of my performance model instance.
- ⋆ **US 3:** As a user, I want to list all queryable performance metrics of the entities contained in the underlying performance meta-model.
- ⋆ **US 4:** As a user, I want to query performance metrics from model entities of my performance model instance.
- ⋆ **US 5:** As a user, I want to aggregate retrieved metrics for further analysis tasks by statistical means.
- ⋆ **US 6:** As a user, I want to access and analyze model revisions, e.g. for analyzing impacts.
- ⋆ **US 7:** As a user, I want to limit structural queries to contain only a specific amount of structural information. *[implied feature]*
- ⋆ **US 8:** As a user, I want to specify model entities to query and reference them with aliases. *[implied feature]*
- ⋆ **US 9:** As a user, I want to control which metrics should be calculated for specific model entities. *[implied feature]*
- ⋆ **US 10:** As a user, I want the performance model instance to be interpreted automatically. *[implied feature]*
- ⋆ **US 11:** As a user, I want to focus on specifying which metrics to compute, but not to parameterize the model solver. *[implied feature]*

Discussion

US 1 allows users to specify which performance meta-model family is required to analyze the model instance and where the model instance can be found. Support for a performance meta-model family is meant as term for a query interpreter that is able to compute metrics for a performance model instance based on a particular meta-model. A more detailed description follows in later chapters.

Afterwards in US 2 and US 3 the user gathers structural information about the model instance and the capabilities of the available model solvers for the specified performance model family. The actual performance query in US 4 then returns performance-relevant results back to the user.

For the US 2 and especially US 3, the user needs to be able to specify, besides of the model instance, model entities. As there might be a large amount of entities in a performance model instance, users shall be able to limit query results using filters as in US 7. For further processing, the user then needs to specify the entities to be analyzed as in US 8. Furthermore US 4 implies that the calculation of metrics can be specified explicitly as in US 9. For analyzing results, e.g. to unveil a resource with least utilization, users need to

assisted through specifying aggregates calculated on top of performance metrics that stem from simulation.

As performance model instances vary in their amount of model entities and structure, users shall be unburdened from interpreting performance model instances manually as in US 10. Furthermore the solving of performance model instances depends on the parameterization of solver components that shall be parameterized automatically as in US 11. Interpretation and parameterization are essential and inseparable steps for the analysis of a performance model instance.

In order to assist a software architect during the composition of components into a software architecture and software developers during the selection of algorithms, the system should offer an interface to access historic instances as in US 6 in order to compare alternatives and to make decisions based on facts.

3.1.2.2. Performance Testing System

Scenario

As a sub-process in Continuous Integration (CI), a system for testing the performance by examining the performance model can be installed. The performance testing system is built to derive automatically a performance model instance from a software system and execute performance queries on this model instance. Deriving a performance model instance might be realized through providing a skeleton of a performance model instance and measuring resource demands by executing several workloads. Using the measurement data, the skeleton can be populated with estimates and performance queries can be used to analyze the impact of changes. Using this technique, performance regressions after complex refactorings, redesigns or upgraded program libraries can be detected. The process is executed by a machine user without human interaction.

Environment

This scenario is executed in an offline environment. The performance testing system is set up during the development phase of a software system. The software development team specifies queries to ensure performance-critical sections of the software system are tested continuously and change impacts can be detected. As the low-level structure of performance model instances might change during the development, high-level entities (e.g. components) will usually remain but their identifiers might vary.

User Stories

- ⋆ **US 12:** As a user, I want the system to resolve model entities by symbolic identifiers independent from the model structure.
- ⋆ **US 13:** As a user, I want to filter model entities by their attributes. *[implied feature]*

Discussion

As in US 4, performance queries are executed on a model instance. Special in this case is the fact, that model instances might be derived automatically and the model structure, e.g. model entity identifiers, can vary. To cope with this problem, users need the capability of queries to list (US 2) and possibly filter (US 13) entities that can be embedded in queries to retrieve metrics. With this capability, machine users can be used to automate the execution and analysis of performance query results.

3.1.3. User Stories for Online Scenarios

3.1.3.1. Performance Analyst

Scenario

A software system has been successfully developed and was deployed on computing resources for operation. The system is serving customer requests and fulfilling business transactions. As the expected workload distribution during design and development time differs from the real-world environment, the deployment of applications shall be optimized based on measurements during operation. Optimization might take place through the new settings of configuration parameters to cope with the workload demands. To control the process a performance analyst queries the performance model instance of the system in operation.

Environment

This scenario represents an online setting. The model instance is parameterized with resource demands and utilization rates from the live system and historic data. Different workload patters, e.g. depending on the weekday or hour of day, need to be considered when making reconfiguration suggestions. Underprovisioning of resources can lead to SLA violations, thus during run-time fast reconfiguration advises are necessary to ensure the QoS. The (recorded) live data is also valuable to make more accurate decisions about a redesign and refactorings for subsequent revisions of the software system.

User Stories

- ⋆ **US 14:** As a user, I want the system to interpret the model instance and list all variable parameters.
- ⋆ **US 15:** As a user, I want to analyze the dynamic behavior of performance models.
- ⋆ **US 16:** As a user, I want to select varying aspects of a performance model for analyzing dynamic behavior.
- ⋆ **US 17:** As a user, I want to control how parameters are varied and define bounds of the parameter space.
- ⋆ **US 18:** As a user, I want to control how my performance model instance parameters are explored.
- ⋆ **US 19:** As a user, I want the model solver to vary DoF for a sensitivity analysis. *[implied feature]*
- ⋆ **US 20:** As a user, I want to specify the least statistical significance level for sensitivity. *[implied feature]*

Discussion

Through the variation of parameter settings for specific entities, i.e. using DoFs, in performance models, a performance analyst can make his advice for a redesign or refactorings of a software system. The advice will be based on a compromise of varying workload profiles. Using the automated parameterization of the model instance, the performance analyst can work efficiently and examine the model dynamics as in US 15. For a goal-oriented analysis of performance models, the performance analyst needs to control the size of and the way the parameter space is explored. The system should assist users and list all variable parameters as in US 14 in order to reduce manual efforts.

If the performance analyst wants to focus on a limited amount of parameters to vary, he needs to be able to select these model entities as in US 16. To further reduce the amount of simulation results, the performance analyst needs to be unburdened from a systematical evaluation of parameter combinations. The system therefore should automatically perform a sensitivity analysis as in US 19 and respect statistical requirements as in US 20. US 18 allows the performance analyst to specify which model exploration strategy to use.

3.1.3.2. Monitoring and Resource Management Systems

Scenario

After a software system has been deployed and transitioned into operation, it is being monitored for availability and other qualitative aspects. A sophisticated monitoring system can make use of a performance model instance and derive trends to recognize and forecast QoS violations before the latter happens. For the predictions, the monitoring system can use the performance model parameterized with live monitoring data obtained during runtime.

By reusing the analyses of the monitoring system, further analyses can be conducted through a resource management system, to control the operation and utilization of all resources.

Environment

This scenario is in an online setting. As the monitoring system is also responsible for alerting staff in case of QoS violations, it has to react within time bounds to evaluate monitoring results. In case of online performance predictions and queries, the system must be able to express the demand for fast, but less accurate, responses.

For a resource management system, that performs reconfiguration tasks to maximize the utilization of resources at the least cost, the demand for accurate or fast predictions varies. The reservation and release of resources depends on the platform provider and therefore the actual demand is defined by the concrete usage scenario.

User Stories

- ⋆ **US 21:** As a user, I want to control the solving of a performance model instance by specifying a trade-off between, e.g. speed and accuracy.
- ⋆ **US 22:** As a user, I want fine grained control of the significance level for stochastic model predictions. *[implied feature]*

Discussion

Using a constraint for the execution of model solvers as in US 21 allows users to advice the system to create responses in a suitable fashion. The actual solving process depends on the model solver, that tries to fulfill the user requirements as good as possible. More fine-grained control is possible through US 22.

3.1.3.3. Application Server or Middleware System

Scenario

An application has been deployed to an application server and is handling customer workload. The application uses various resources provided by the application server, e.g. thread pools and connection pools. Furthermore the application server takes care of the memory management and garbage collection. The configuration of the resources is done by system operators during deployment time.

Environment

This scenario is in an online environment. The application is serving customer requests and the application server provides exclusive resources to the application. The resources have tunable parameters regarding the size (e.g. for thread pools) or frequency parameters (e.g. garbage collection). For highest efficiency, these parameters need to be adjusted to the current workload.

User Stories

- ⋆ **US 23:** As a user, I want to explore possible bottlenecks in applications.
- ⋆ **US 24:** As a user, I want to detect the abundance of resources in allocations.
- ⋆ **US 25:** As a user, I want to derive application performance models during run-time. *[implied feature]*
- ⋆ **US 26:** As a user, I want to control the automated derivation of performance models with focus on specific parts. *[implied feature]*

Discussion

Besides of queries centering on raw performance metrics, querying performance models for optimization advice should be possible. In US 23 and US 24 the user triggers the automated analysis of a model instance. Given these information, the user can start to alter tunable parameters at run-time to achieve an efficient execution state.

As reconfiguration actions change the model structure and parameter values, US 25 motivates the creation of a facility for deriving model instances during run-time. In addition in US 26 the granularity of the derived model is focused to allow to derive more abstract model instances. Using more abstract performance model instances, the effort for simulation can be reduced. For both user stories the requirement is exposed to performance meta-model connectors, as from the language part these requirements are too specialized for providing generic language constructs.

3.2. Classification Approach for Online Performance Queries

3.2.1. Overview of the Classification Approach

Based on the usage scenarios and user stories from the previous section, the classification scheme introduced in this section will present a structured overview of our approach for different classes of performance queries. The differences between the query classes manifest in their functionality and expressiveness. In Table 3.2 the classes with a brief description of their intention are shown.

The classification scheme allows to define an order of query classes. Each class has at least one expression consisting of one or multiple keywords associated, i.e. a way to express an information demand, that differentiates it from other classes. When a new class provides a new expression and furthermore leverages an expression from another class, it is meant to be a higher-order class. Applying this ordering to the classes as shown in Table 3.2, it is possible to derive a hierarchy of query classes (i.e. layers) and furthermore expressions can be reused in superior classes. Another illustration of the query class hierarchy would be to interpret the language constructs provided by a class as components that can be reused by components from other, higher-order, classes.

As described in Section 3.1, the functional requirements differ among user roles. As the user stories are intended to describe objectives for the design and need only be specified

Query Class	Query Type	Expression	Description
Model Access	Model Instance Access	`USING`	Access a model instance, i.e. load it from a filesystem
	Model Entity Access	`FOR`	Access a model entity by interpreting the model instance
Model Structure	Listing of Entities	`LIST ENTITIES`	List all analyzable entities and optionally filter by a condition
	Listing of Metrics	`LIST METRICS`	List all calculable metrics for specified entities
Performance Metrics	Basic Query	`SELECT`	Calculate requested performance metrics for specified entities
	Constrained Query	`CONSTRAINED AS`	Calculate metrics with constraints regarding the calculation process
	DoF Query	`EVALUATE DOF`	Analyze performance metrics under varying parameterization of the model instance
Performance Issues	Detect Bottlenecks	`DETECT BOTTLENECKS`	Detect bottlenecks in performance model instances

Table 3.2.: Query Classification Matrix

once, the given collection of user stories is not complete for each role. It implies that objectives and functionality can be reused on demand by across different roles. The classes to be introduced are therefore part of the different user roles, but provide a summary of similar functionality. A description of the classes follows in Section 3.2.2 with references to grammar rules of the resulting DQL.

3.2.2. Objectives for different Query Classes

Model Access

This query class provides expressions for technical demands while accessing performance model instances. A technical demand can be a way to express to load a specific performance model instance from its serialized form in a file system, which is expressed in US 1. After loading the model instance, a user has to handle the entities and therefore a way to name the relevant model entities is needed as defined by US 8.

To summarize, this class is intended to provide expressions to enable users to *access model instances* and to *access model entities*, where access is meant to load and reference model instances or model entities.

- **Query Class:** Model Access
  - ⇝ Model Instance Access: Load and Instantiate, see UsingClause
  - ⇝ Model Entity Access: Reference and Use, see ForClause

Model Structure

Using queries from this class, users can discover elements from a performance model instance that are either interpreted as resources, services or metrics. Metrics are associated to an entity, i.e. either a resource or service. This class follows US 2 and US 3 and their implied user stories.

An integral part of this query class is the demand for US 10 as the interpretation of model element types depends on a performance meta-model family connector. The language

itself shall not make any assumptions about model element types, but leverage information supplied by the connector. This ensures a loosely coupled and reusable language design.

The expressions provided by this class are essential to allow users to rely on the query language solely. A user can discover model elements and is not forced to interpret the low-level characteristics of the performance meta-model family. This query class enables users to *list analyzable model entities* consisting of *listings of resources and services* together with their associated *metrics*.

- **Model Structure**
  - ⇝ Listing of Entities: Interpret and List, see ListEntitiesQuery
  - ⇝ Listing of Resources & Services: Interpret, List and Filter, see ListResources-Query
  - ⇝ Listing of Metrics: List and Derive, see ListMetricsQuery

Performance Metrics

This class supports users to derive performance metrics from performance model instances as of US 4. As for the preceding class, the user is neither faced with low-level details of the performance meta-model family, nor with further solver properties for configuration purposes. To satisfy demands arising from the usage in online scenarios as in US 21, users shall be able to control the level of details in simulations and accuracy with a solver constraint.

To cope with questions arising from dynamic aspects in performance models, i.e. DoFs for model entities, this class offers facilities to configure the exploration and analysis of DoFs as in US 15 and related user stories. The semantics of queries in this class depend on the underlying performance meta-model family and are therefore out of scope for the language design.

Expressions of this class serve for the extraction of performance metrics from performance model instances without low-level model knowledge. Queries in this class allow users to *select metrics* for model entities, *constraint* the execution of the solver and allow the *exploration of DoF*. Facilities for solving DoF increase the productivity of users as they can partly automate the process of analyzing dynamic behavior in performance model instances.

- **Performance Metrics**
  - ⇝ Basic Queries: Select Metrics and Request Solving, see SelectQuery
  - ⇝ Constrained Queries: Constrain Model Solver Execution, see ConstraintClause
  - ⇝ DoF Queries: Explore DoF in Performance Model Instances, see DoFClause

Performance Issues

Instead of interpreting query result manually, this query class is intended to perform an automated DoF exploration to detect shortcomings in the allocation and design of resources and services. These shortcomings might result in bottlenecks, limiting the overall performance as stated for US 23.

Within this class the user is only required to define the performance model instance and the DoF. Then a query is formed of keywords to *detect bottlenecks* under varying usage scenarios.

- **Performance Issues**

  ⇝ Bottleneck Detection: Automated Bottleneck Search, see DetectBottlenecks-Query

Optimization Problems

As highest-order query class, this class is intended for the formulation of optimization problems. By the term optimization problem a user defines the targeted result of a performance analysis in a well-structured fashion. As this class is out of scope of this thesis and depending on the results of underlying classes, we do not make assumptions on language constructs for this class.

3.3. Specification of Functionality provided by Query Classes

3.3.1. Overview of Query Class Functionality

In Section 3.3.2 access to model files is described followed by facilities to access the structure of model entities in Section 3.3.3. The next layer for user-controlled performance analyses in Section 3.3.4 offers functionality to express performance-related queries to performance models delivering performance metrics. The functionality for these layers is summarized in Figure 3.1a.

Upon this basic model solving functionality, further performance issue-related approaches are introduced in Section 3.3.5 and an outlook to optimization problem-related queries is given in Section 3.3.6. These layers are summarized in Figure 3.1b.

3.3.2. Functionality for Model Access

This part of the query language deals with loading and accessing performance model instances. Furthermore, this query class also allows to access entities in the performance model instance to create an alias for them. Expressions from this class are based on the expression `USING` to specify a performance model instance and `FOR` to name relevant entities from the performance model instance.

The `USING` Expression

As the query language is not limited to a single performance meta-model and the persistence mechanisms might differ significantly between performance meta-models, the loading of models is delegated to a performance meta-model specific plug-in parameterized with a suitable location identifier, e.g. a namespace identifier for the plug-in concatenated with a Uniform Resource Identifier (URI).

The `FOR` Expression

By using the `FOR` expression, users can specify which performance model entities are relevant for the analysis. Only entities specified in this expression can be used to retrieve performance metrics from. Additionally, `FOR` supports the aliasing of model entities (i.e. services or resources) to assist users working with the query language. The identifiers of model entities depend on the underlying performance meta-model, that might rely on anything ranging from subsequent integer-based identifiers up to Universally Unique Identifiers (UUIDs). As for further operations on the entities, the user would have to repeat the identifier, the query might suffer from illegibility. To circumvent this shortcoming, a central alias mapping is necessary.

3.3.3. Querying the Model Structure

Queries for the model structure are intended to serve information demands of users. Support for these queries is necessary to provide a unified interface for the analysis of performance model instances. Without this query class additional tools depending on the performance meta-model family would be necessary to work with the performance model instance. Using the structural information, performance queries can be formulated and executed. The support and interpretation of the model structure is contained in a plug-in that is aware of a specific performance meta-model family.

To query the model structure, the principal keyword for this query class is `LIST`. To gather information about queryable model entities the `LIST ENTITIES` expression can be used. After entities have been discovered, the plug-in can be queried for available metrics of the entities using `LIST METRICS` queries. The use of these queries allows users to build higher-order queries afterwards and to start to analyze the performance model.

The `LIST ENTITIES` Expression

As different performance meta-models are supported by the query language, no general assumption about the queryable model entities is possible. To offer a general interface, the query language distinguishes only between *services*, i.e. model entities processing a workload, and *resources*, i.e. model entities utilized by services to handle workloads.

The actual decision whether a model entity is a service or a resource, thus being an entity providing performance metrics, depends on the connector for the performance meta-model. To cope with this problem, the plug-ins are provided with a meta-model for mapping their service- and resource-like entities to the query language. This mapping approach allows to abstract from specific performance meta-models while retaining all structural information required for performance queries.

In addition to `LIST ENTITIES` the expressions `LIST RESOURCES` and `LIST SERVICES` exist. These query types are derived from `LIST ENTITIES` with implicit filtering. The implicit filter ensures the result of the query will only contain resources respectively services from the performance model instance.

The `LIST METRICS` Expression

As the interpretation of services and resources might vary between performance meta-models, the metrics computable for a given model entity might also vary. Therefore, the `LIST METRICS` expression queries the plug-in for the performance meta-model family, as specified in the `USING` expression, which responds with available performance metrics for a given model entity.

Concluding the keywords `LIST ENTITIES` and `LIST METRICS` unburden users from analyzing the model structure manually and model-specific knowledge how the model can be transformed to retrieve metrics.

3.3.4. Querying Performance Metrics in Model Instances

With the ability of structural model access through a performance meta-model plug-in, the next layer of query classes is related to queries for performance metrics. The `SELECT` keyword is the principal keyword for this query class. To form a `SELECT` expression for a performance analysis, the following workflow is suggested.

The `SELECT` Expression

To form a query for analyzing performance metrics through the `SELECT` expression, the following steps are necessary:

1. Define which performance model instance to use ⇒ `USING`
2. Define which model entities shall be analyzed ⇒ `FOR`
3. Define which metrics shall be computed ⇒ `SELECT`

The combination of these language constructs allows to retrieve performance metrics for specific model entities. The availability of metrics depends on the evaluation of the used performance model instance and further run-time aspects. As these properties depend on the performance meta-model family, their handling and evaluation is part of a specific plug-in and not in scope for the classification scheme.

As in Figure 3.1a shown, the family of *Basic Queries* is the first layer in queries for performance metrics. The following layers, i.e. *Constrained Queries* and *DoF Queries*, will serve for more complex query scenarios.

The `CONSTRAINED AS` Expression

For online performance queries, the response time of the computation of performance metrics can be crucial to fulfill subsequent tasks. The objective of further tasks might involve triggering reconfiguration processes of the resource landscape to cope with SLA requirements. From a language perspective, involving the query language and the execution logic of the query language, only an advice for the computation of performance metrics can be made. The keyword forming the expression for this advice is `CONSTRAINED AS`.

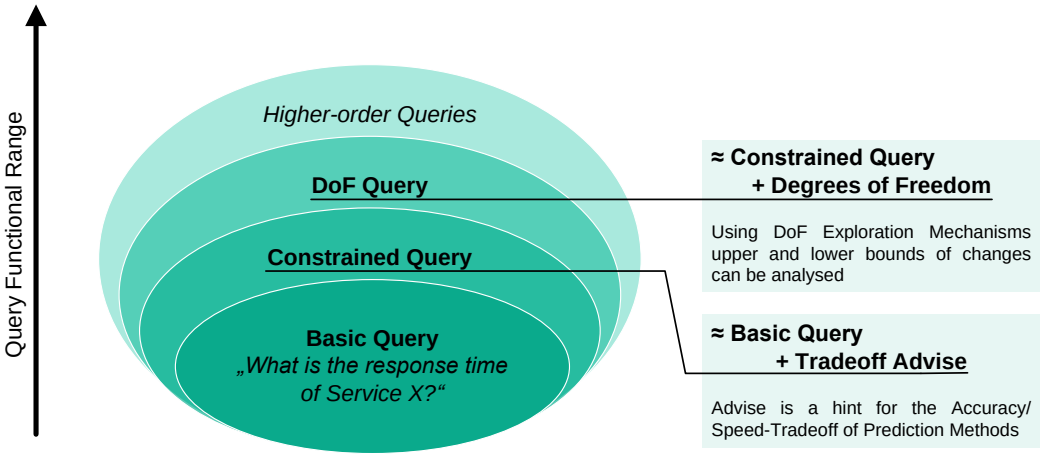
From a plug-in perspective, there might be several points how to handle the advice. A first target for the trade-off might be the model transformation from the originating performance model into the simulation or analytic model. The model instances for performance metric calculation might be varied in their level of granularity, where less simulation model artifacts might mean less computation effort. Plug-in developers may therefore consider the advice specified through the `CONSTRAINED AS` expression, the referenced entities in the `FOR` expression and additionally the requested metrics in the `SELECT` expression to realize tailored model transformations.

For model solvers based on statistical modeling or statistical processes, the advice can be interpreted to stop all computations at a given level of statistical significance of the computed result and to return the result. Depending on the actual performance meta-model and the metrics to be calculated there might even be more decision points for handling the trade-off advice.

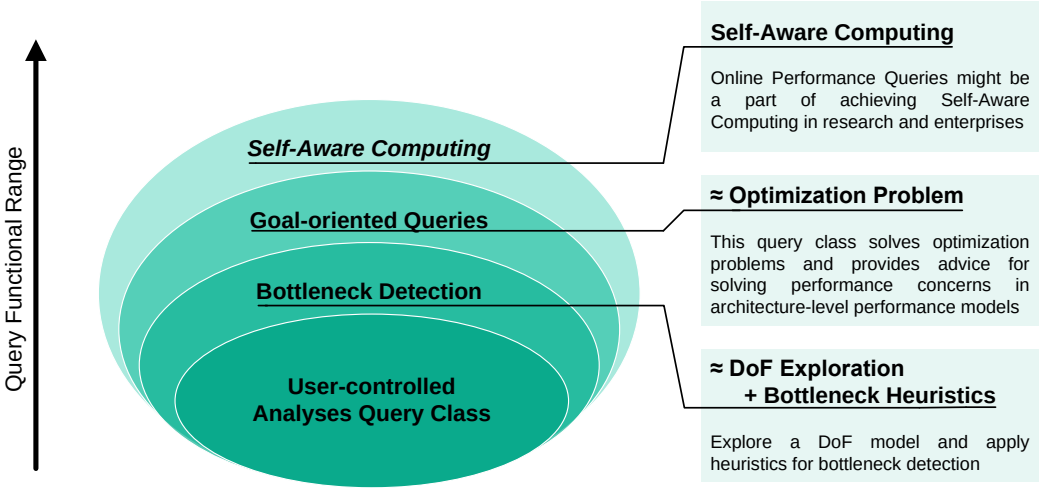
As the availability of solver constraints is influenced by several factors and depends directly on the components used for solving DQL queries, the query language makes no assumption on the available solver constraints. Each performance meta-model specific plug-in must therefore provide descriptions of available constraints.

The `EXPLORE DOF` Expression

The next higher layer for performance queries as in Figure 3.1a is the exploration of DoFs in a performance model instance. Using DoFs in performance model instances allows to express variability, e.g. in the resource layer where containers of resources might be provisioned with varying amounts of resources. If a computing system is modeled with these DoFs, the key question is to find an efficient configuration, while taking the QoS



(a) Layers of Queries for User-Controlled Analyses



(b) Layers of Queries for Automated Performance Model Solving

Figure 3.1.: Overview of the Query Classification Approach

into account. In Section 4.2 the current state of the art in modeling and solving DoF is outlined. The expression for DoF exploration is based on the `EXPLORE DOF` keyword, requiring several subsequent statements to configure the exploration optionally.

For the exploration of a model instance to find parametric dependencies in tuning parameters, e.g. the configuration settings of the resource landscape, the exploration needs to perform a partitioning of the parameter space and a sensitivity analysis while performing parameter variations. Therefore the language will offer configuration options as the expression `GUIDED BY` to select an exploration strategy from an exploration strategy-aware plug-in and the accuracy trade-off for the sensitivity analyses using the expression `CONSTRAINED AS` as used for constrained queries. Finally, to cope with performance models missing an integrated modeling of DoF, a user can supply a DoF model specification and customized plug-in through the expression `WITH`.

To foster the usability of DoF exploration and to allow comprehensive DoF models, a user can select which parameters to vary using the `VARYING` expression. Especially in online scenarios, exploring a full parameter space would impose a high computation effort. Limiting the parameter space is therefore crucial for DoF exploration in online scenarios and supports tailored model transformations as irrelevant DoFs can be sliced out from analytical or simulation models.

Although a manual exploration of DoF is possible, the exploration in complex models is time-intensive, due to manual efforts for parameter variation subject to inaccuracy and error-prone. As this class deals with user-controlled analyses, drawing conclusions from explorations is not part of the language and its execution. Users have to decide which model parts to vary and how to steer the performance analyses in order to extract relevant performance metrics.

3.3.5. Querying and Analyzing Performance Issues

Figure 3.1b shows the relevant classification for *Higher-order Queries* as shown in Figure 3.1a. Although these classes are out of scope for the implementation of the query language in this thesis, the design of queries for performance issues shall be outlined. As a representative example for performance issue detection approaches, queries for bottleneck detection will be outlined in this section. A brief overview about the state of the art in bottleneck detection can be found in Section 4.4.

Bottlenecks in performance models are resources limiting the system's total performance because of saturation under a specific workload. In real-world scenarios bottlenecks can degrade the performance of systems and lead to SLA violations. An example for a bottleneck in real-world environments, that could be found through the analysis of a performance model instance, is the provisioning of Random Access Memory (RAM) for a VM. If the provisioned amount of RAM is not sufficient, swapping of memory pages is necessary in order to complete customer requests.

The additional resource demand for swapping and the resulting higher response times could cause SLA violations. Using a fine-grained performance model instance, this bottleneck could have been detected through analyses of workload resource demands and their effects on the resource landscape.

The `DETECT BOTTLENECKS` Expression

The class of queries for performance issue analyses is represented through the keyword `DETECT`. To perform a bottleneck analysis, queries using the `DETECT BOTTLENECK` expression can be used. The bottleneck detection shall reuse functionality for DoF exploration.

Using varying usage profiles or varying resource allocations, bottlenecks might be detected through sensitivity analysis and optimized detection heuristics.

Another query type from this class might be a mechanism for detecting overprovisioning of resources. The functionality of this kind of query might be related to bottleneck detection, but with exploration strategies working in opposite direction (e.g. *minimize allocation until saturation with fixed workload* vs. *maximize resource utilization until saturation with varying workloads*). Further analysis on this query type should unveil whether this type would fit into a query class and which objectives need to be achieved for providing an implementation.

3.3.6. Optimization Problem Queries

Based on the previously introduced query classes, the query class for *Goal-oriented Queries* shifts from analyzing performance model instances manually towards an entirely automated approach to solve optimization problems for efficient allocation and operation. This class combines approaches from several other classes, e.g. reusing bottleneck detection heuristics as in Section 3.3.5, but with an automated configuration of DoFs to be varied. In [BHK12] an example for an optimization problem is presented as: *"How should the system configuration be adapted to avoid QoS problems or inefficient resource usage arising from changing customer workload?"*.

Currently the language design does not contain language constructs to form expressions of optimization problems. To design and implement these expressions, a combination with the S/T/A Meta-Modeling approach could be a reasonable foundation [HBK12]. The S/T/A Meta-Modeling allows to model reusable adaptations applied to performance model instances in order to enable an automation of management processes for efficient operation of software systems. Further research on the S/T/A Meta-Modeling could include a linguistic approach for modeling optimization problems in order to link them with adaptation strategies.

3.4. Evaluation of User Stories and Usage Scenarios

3.4.1. Introduction to the Evaluation of Expressions

The previous sections introduced user stories, see Section 3.1, an approach for classification of queries, see Section 3.2, and finally the necessary functionality for the different query classes is introduced in Section 3.3. To discuss the design approach of DQL as query language for online performance queries, this section will evaluate exemplary queries and link queries to their originating user stories from Section 3.1. A more formal description of language constructs provided by DQL and rules for creating expressions will follow in Section 3.5, where the resulting grammar rules are explained in detail.

The ordering of query examples and their discussion will follow a bottom-up workflow approach of using DQL in a performance analysis. The starting point of the performance analysis is an architecture-level performance model and a simulation engine capable of handling online and offline queries. The designated workflow for the performance analysis is defined as:

0. *Preparation:* Create/provide a performance model instance
1. Explore the structure of the performance model instance
2. Analyze available performance metrics
3. Request the computation of performance metrics

In addition advanced examples are presented capturing the analysis of DoFs in Section 3.4.4 and how to access historic performance models and to retrieve performance metrics from these instances in Section 3.4.5. The evaluation of the query examples given in this section is conceptually. In Chapter 6 implementation approaches will be evaluated by executing queries through DQL and its implemented software components connected to a real performance model instance and with its related toolchain.

The class of Performance Issue Queries is not evaluated within this section or further chapters as this class is at the current state of this thesis not yet a fully implemented part of DQL.

3.4.2. Evaluation of Model Structure Queries

Listing 3.1 shows the entry point for performance analyses using an architecture-level performance model (US 2). The query returns an instance of the Mapping Meta-Model. The Mapping Meta-Model is a data structure for the abstraction from a given performance meta-model family to the means of DQL as described in Section 5.2. The result contains all available entities, i.e. services and resources, found in the model instance queried.

The entities are annotated with an absolute identifier represented as `java.lang.String` and, if the DQL Connector can supply this information, a human-readable alias is set as suggestion for an alias in further queries (US 12). A DQL Connector is named through the `USING` expression and is a software component capable of handling a specific performance meta-model family (US 1). The `USING` expression is two-fold and contains also the location of the performance model instance requested. The parts are separated by the `@` character.

```
1 LIST ENTITIES
2 USING nop@'void://url';
```

Listing 3.1: Example of a Model Structure Query with `LIST ENTITIES`

The actual selection and creation of an instance of an adequate DQL Connector is delegated to the DQL Query Execution Engine (QEE), which is a software component responsible for executing the query. Detailed descriptions of DQL Connectors and the DQL QEE will be given in Chapter 5.

The next query in Listing 3.2 shows an example to retrieve information on the available performance metrics that can be computed for entities of interest (US 3). As the computability of performance metrics can depend on conditions that can not be defined at a language level, static information about metrics can not be provided by the means of DQL.

```
1 LIST METRICS (RESOURCE 'id1' AS res1,
2                SERVICE  'id2' AS svc1)
3 USING nop@'void://url';
```

Listing 3.2: Example of a Model Structure Query with `LIST METRICS`

Given the information gathered through the queries shown in Listing 3.1 and Listing 3.2, a user is able to proceed to request performance metrics through DQL.

3.4.3. Evaluation of Performance Metrics Queries

This section proceeds to conduct a performance analysis. Listing 3.3 shows an example of a query for performance metrics with a constraint, i.e. a *Constrained Query* (US 4, US 21).

The query requests the computation of two metrics for two different model entities. The results are represented as instance of the Mapping Meta-Model and, if applicable, metrics which failed to be calculated, are indicated as invalid result.

```
1 SELECT res1.metric1, svc1.metric2
2 CONSTRAINED AS "accurate"
3 FOR RESOURCE 'id1' AS res1,
4     SERVICE  'id2' AS svc1
5 USING nop@'void://url';
```

Listing 3.3: Example of a Performance Metrics Query with `SELECT`

As analysis tasks might require to compute statistical metrics on top of performance metrics, Listing 3.4 gives an example for the computation of an aggregate (US 5). In this case, the computation is expected to work by a best-effort approach, i.e. to compute the aggregate for those entities, where the requested metric available and valid. Error messages during the computation will point out missing metrics and the result will be marked as invalid.

```
1 SELECT PERCENTILE(*.utilization) [ percentile="90" ]
2 FOR RESOURCE 'id1' AS blade1, RESOURCE 'id2' AS blade2,
3     RESOURCE 'id3' AS blade3, RESOURCE 'id4' AS blade4
4 USING nop@'void://url';
```

Listing 3.4: Example of a Performance Metrics Query with `SELECT` and an Aggregation Operation of Results

Using the queries from Section 3.4.2 followed by the examples from this section, a user is able to perform analysis tasks without the need for additional tools. DQL integrates the necessary support for auxiliary tasks like browsing for entities, testing for available metrics, controlling a solver and computing aggregated statistical metrics.

3.4.4. Evaluation of Degrees-of-Freedom in Queries

As an advanced scenario of performance analysis, DQL allows to analyze dynamics through controlling the simulation of DoFs (US 15). To support users in these analyses, DQL allows to query all available DoFs as shown in Listing 3.5 (US 14). The concept is similar to the concepts of the Model Structure Query class as shown in Section 3.4.2. At the current implementation state, querying the allowed bounds of the parameter space of a DoF is not yet implemented at the language level.

```
1 LIST DOF
2 USING nop@'void://url';
```

Listing 3.5: Example of a Model Structure Query with `LIST DOF`

Listing 3.6 shows a query for performance metrics, as in Listing 3.3, but with the extension to explore DoFs. Through the keyword `EVALUATE DOF` it is extended to analyze DoFs (US 16). For each DoF, the parameter space is defined through iterating numerical values. The different syntax representations for the parameter space definition will be explained in Section 3.5.8. The exploration of the parameter space is controlled through an exploration strategy (US 17, US 18). The result of the query is one instance of the Mapping Meta-Model for each valid combination of DoFs in the parameter space.

```
1 SELECT res1.metric1, svc1.metric2
2 CONSTRAINED AS "accurate"
3 EVALUATE DOF
4  VARYING 'idDoF1' AS DoF1 <1 .. 100 BY 1>,
5          'idDoF2' AS DoF2 <1, 2, 3, 4>
6  GUIDED BY "FullParameterSpaceExploration"
7 FOR RESOURCE 'id1' AS res1,
8     SERVICE  'id2' AS svc1
9 USING nop@'void://url';
```

Listing 3.6: Example of a Performance Metrics Query with `SELECT` and `EVALUATE DOF`

Based on the foundations shown in Section 3.4.2 and Section 3.4.3, this section demonstrates the reusability of concepts for composing advanced queries capturing performance model dynamics. This way, once a user has put effort in building a Performance Metrics Query, this effort can be leveraged to build queries for analysis tasks on top of first insights.

3.4.5. Evaluation of a Temporal Dimension in Queries

For analyzing model instance changes or their evolving over time, to aggregate historic information or to access a historic model instance, a user can query model-specific time units through a statement like Listing 3.7 (US 6). As in Section 3.4.4 the query is an extended form[1] of the example shown in Section 3.4.3. The interpretation of the time unit and the aggregation of samples is specific for the given performance meta-model and the underlying DQL Connector. The language does not provide any static information of time units.

```
1 SELECT res1.metric1, svc1.metric2
2 FOR RESOURCE 'id1' AS res1,
3     SERVICE  'id2' AS svc1
4 USING nop@'void://url'
5 OBSERVE 48 ModelInstances SAMPLED BY 1h;
```

Listing 3.7: Example of a Model Structure Query with `SELECT` with `OBSERVE` using custom Time Units

Listing 3.8 shows another example for the specification of time units. These time units allow to define a time frame that is in line with the clock. The interpretation of these time units is therefore defined by the means of the implementation of DQL. For relative time specifications, the interpretation is explicitly defined in Section 5.3.5. Having multiple options, i.e. absolute and relative time specifications in addition to specific time units, users are free to decide how to access model instances and revisions.

```
1 SELECT res1.metric1, svc1.metric2
2 FOR RESOURCE 'id1' AS res1,
3     SERVICE  'id2' AS svc1
4 USING nop@'void://url'
5 OBSERVE BETWEEN '2012-01-01T10:00' AND -3m 2d 4h;
```

Listing 3.8: Example of a Model Structure Query with `SELECT` with `OBSERVE` using `BETWEEN`

[1] The `CONSTRAINED AS` keyword is omitted as it does not foster the expressiveness.

Besides of performance analysis during design and development phases, DQL offers support for time units allowing to analyze historic performance data. The support of these concepts fosters the claim of DQL to offer an unified interface to access performance model instances and to eliminate the need for auxiliary tools or manual efforts.

3.4.6. Summary

The query examples have shown that a performance analysis of a performance model instance through DQL is possible. A user can rely on the language constructs offered and is not forced to use assisting tools. Through DQL the toolchain required for solving, transforming and validating performance model instances is hidden.

As an advantage of a textual syntax for queries, DQL allows to extend expressions through additional keywords with a low effort. Using additional keywords new language functionality can be added to the existing language structure, while previously defined queries can still be used. Users are therefore neither forced to change their existing queries when new functionality is added, nor are forced to learn new features continuously.

An example for this advantage is using the `EVALUATE DOF` expression to extend a Performance Metrics Query towards analyzing DoFs or using the `OBSERVE` expression for adding a temporal dimension to queries as shown in the previous sections.

3.5. Query Language Rules and Terminals

3.5.1. Conventions and Basic Grammar Rules

3.5.1.1. Conventions

The subsequent sections will introduce the grammar rules defining the DQL to perform online performance queries. The grammar rules are visualized through syntax diagrams[2]. To distinguish between technical terms, a non-terminal rule is represented as NonTerminal and a terminal rule as `Terminal`. These typographic conventions for grammar rules will be used in the remaining document. An index of referenced grammar rules can be found in Appendix B.

3.5.1.2. Rule ID

The non-terminal ID is provided by Xtext [Ecl12]. It is based on the Identifier non-terminal from [GJS+11, p. 23 ff.]. The non-terminal can be used, e.g. to represent Java class or method names.

3.5.1.3. Rule INT

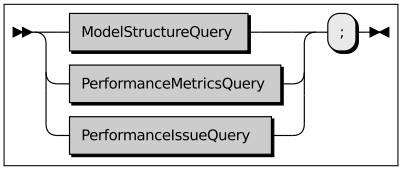
The non-terminal INT is provided by Xtext [Ecl12]. It is based on the DecimalNumeral non-terminal from [GJS+11, p. 25 ff.]. The non-terminal can be used to represent decimal numbers consisting of one or multiple digits.

3.5.1.4. Rule STRING

The non-terminal STRING is provided by Xtext [Ecl12]. It is based on the StringLiteral non-terminal from [GJS+11, p. 36 ff.]. The non-terminal can be used to represent instances of `java.lang.String` found in Java source code.

[2] The graphical syntax is based on the Railroad Diagram Generator at `http://railroad.my28msec.com/rr/ui`.

3.5.1.5. Rule DescartesQL

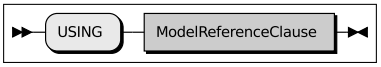


DescartesQL is the foundation for all valid statements that can be expressed through DQL. The rule allows to choose between the non-terminal rules for queries from the different query classes and is finished by a terminal ;.

See Section 3.2, ModelStructureQuery, PerformanceMetricsQuery and PerformanceIssueQuery for more information.

3.5.2. Grammar Rules for Model Access

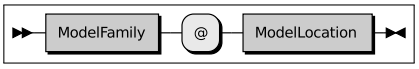
3.5.2.1. Rule UsingClause



The UsingClause provides information for accessing a model instance through a DQL Connector and is essential to Model Structure Queries, Performance Metrics Queries and Performance Issue Queries. In ModelReferenceClause all information for referencing a model instance is stored.

See Section 3.3.2 and ModelReferenceClause for more information.

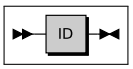
3.5.2.2. Rule ModelReferenceClause



A ModelReferenceClause contains the information required to start executing a query. To delegate a query to a DQL Connector, the DQL QEE is used to look up a suitable DQL Connector with the information provided through ModelFamily. The ModelLocation contains a Connector-specific value referencing a model location.

See Section 5.1, ModelFamily and ModelLocation for more information.

3.5.2.3. Rule ModelFamily



A ModelFamily is specified as ID and used to identify a DQL Connector for executing the query.

3.5.2.4. Rule ModelLocation

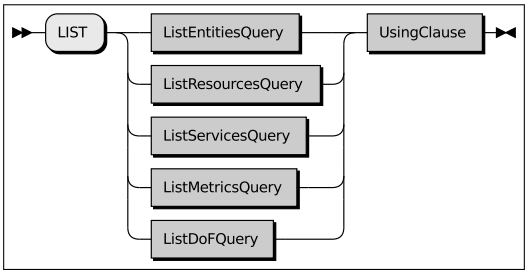
A ModelLocation is specified as STRING and used to specify where a model instance can be found.

3.5.3. Grammar Rules for Model Structure Queries

3.5.3.1. Rule ModelStructureQuery

In ModelStructureQuery the top-level queries for the class of Model Structure Queries are aggregated. Currently the class is solely based on the rule ListQuery.

3.5.3.2. Rule ListQuery

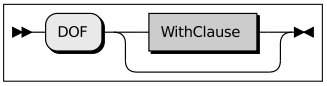


ListQuery is a compilation of different queries to list entities from a model instance. Queries from this class can be recognized through starting with the keyword `LIST` and utilizing an UsingClause to reference a model instance to be analyzed.

3.5.3.3. Rule ListEntitiesQuery

ListEntitiesQuery is a supertype of ListResourcesQuery and ListServicesQuery to retrieve a listing of entities with performance-centric properties, e.g. resources or services. Subsequently ListMetricsQuery can be used to retrieve available performance metrics for entities.

3.5.3.4. Rule ListDoFQuery



The ListDoFQuery is a specialization of the ListQuery and is used to list all DoFs found in a model instance. Through a DoF, properties of the model instance can be altered reflecting changes in made to a system to be analyzed, e.g. different input parameters for algorithms or more computing resources.

See Section 3.3.4 and WithClause for more information.

3.5.3.5. Rule ListResourcesQuery

The ListResourcesQuery is a specialization of a ListEntitiesQuery containing only a listing of resources. See ListEntitiesQuery for more information.

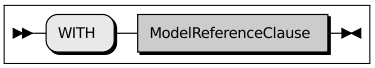
3.5.3.6. Rule ListServicesQuery

The ListServicesQuery is a specialization of a ListEntitiesQuery containing only a listing of services. See ListEntitiesQuery for more information.

3.5.3.7. Rule ListMetricsQuery

A ListMetricsQuery determines all computable metrics for entities contained in the EntityReferenceClause. As the availability of computable metrics depends on conditions that are not definable at a language level, the DQL Connector provides these metrics. This approach fosters flexibility in implementing DQL Connectors.

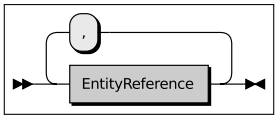
3.5.3.8. Rule WithClause



Through a WithClause an additional information source for DoF models can be used. The structure of this clause is similar to the UsingClause, but using `WITH` as initiating terminal. The interpretation of the additional information source is subject to the DQL Connector referenced in the UsingClause.

See UsingClause for more information.

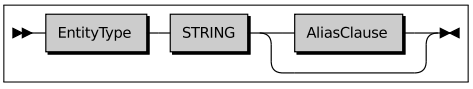
3.5.3.9. Rule EntityReferenceClause



The EntityReferenceClause consists of one or more instances of the rule EntityReference to select entities from the model instance referenced through the UsingClause.

See EntityReference and UsingClause for more information.

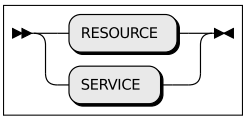
3.5.3.10. Rule EntityReference



An EntityReference is used to reference an entity found in a performance model instance. The performance model instance is referenced by a UsingClause. The rule consists of EntityType to specify the entity type, a STRING to specify the absolute identifier of the entity in the source model instance and an optional AliasClause to specify a more convenient alias, especially is case of entity identifiers that are generated automatically and not optimized for readability.

See UsingClause, EntityType and AliasClause for more information.

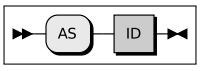
3.5.3.11. Rule EntityType



The rule EntityType consists of terminals for all entity types that are supported in Performance Metric Queries, i.e. entities that are directly queryable. Through this rule, the set of allowed entities in queries limited at the language level.

See Section 3.2 for more information.

3.5.3.12. Rule AliasClause



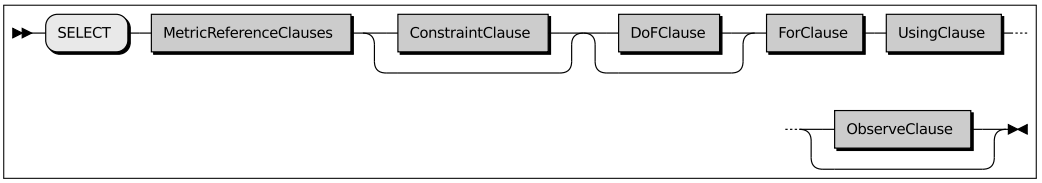
The AliasClause allows to specify a user-friendly identifier as alias for an entity. The alias can be used in other parts of the query to reference the aliased entity from the model instance specified in the UsingClause. An alias can be specified through the terminal `AS` followed by the alias specified as ID.

3.5.4. Grammar Rules for Performance Metrics Queries

3.5.4.1. Rule PerformanceMetricsQuery

In PerformanceMetricsQuery the top-level queries for the class of Performance Metrics Queries are aggregated. Currently the class is solely based on the rule SelectQuery.

3.5.4.2. Rule SelectQuery



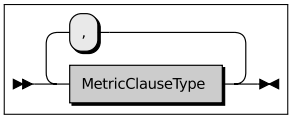
The SelectQuery consists of mandatory an optional parts. The mandatory parts are the foundation for executing a query to retrieve performance metrics. The optional parts extend the functionality towards higher query classes and in case of the ObserveClause it extends the functionality of model access.

Mandatory parts of a SelectQuery in a bottom-up way are the UsingClause to reference a model instance, the ForClause to specify which model entities are relevant for the query, MetricReferenceClauses to specify the performance metrics to be computed for the relevant entities and the terminal `SELECT`.

The optional rules are the ConstraintClause to extend the query towards the Constrained Query Class and the DoFClause, see Section 3.5.7, to extend the query towards the DoF Query class, see Section 3.5.8. Through an additional temporal dimension specified by the ObserveClause, see Section 3.5.6, the way the model instances are accessed is changed to allow to query other performance model instance revisions.

See Section 3.3.4, Section 3.4.3, Section 3.4.4, Section 3.4.5 for more information. Relevant clauses are following.

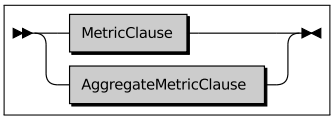
3.5.4.3. Rule MetricReferenceClauses



MetricReferenceClauses consist of one or more instances of the rule MetricClauseType to reference entities found in the ForClause and metrics computable for these entities.

See ListMetricsQuery and ForClause for more information.

3.5.4.4. Rule MetricClauseType



Within MetricReferenceClauses two different types of metric-referencing rule instances are valid. A user might either choose a plain MetricClause to reference an available performance metric directly or an AggregateMetricClause to compute an aggregate of performance metrics.

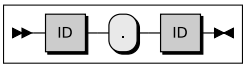
See MetricClause, Section 3.4.3 and AggregateMetricClause for more information.

3.5.4.5. Rule MetricClause

A MetricClause contains, through an instance of the rule MetricReference, a direct reference to a performance metric computable for an entity.

See MetricReference for more information.

3.5.4.6. Rule MetricReference



The MetricReference is provided by an instance of ID to provide an identifier or alias for the entity, a terminal . to separate the entity referencing identifier or alias from the metric name and another instance of ID to provide the metric name.

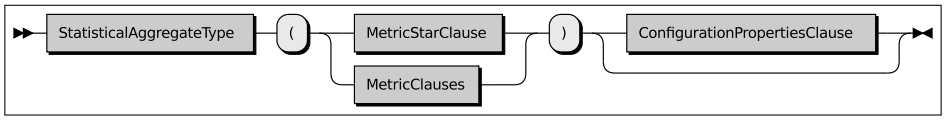
3.5.4.7. Rule ForClause

In a ForClause all entities being used in a query, are listed through EntityReferenceClause. Entities not listed within this clause are therefore not usable, but might be used implicitly through the DQL Connector, e.g. for calculating other performance metrics.

See EntityReferenceClause for more information.

3.5.5. Grammar Rules for Aggregate Calculation

3.5.5.1. Rule AggregateMetricClause

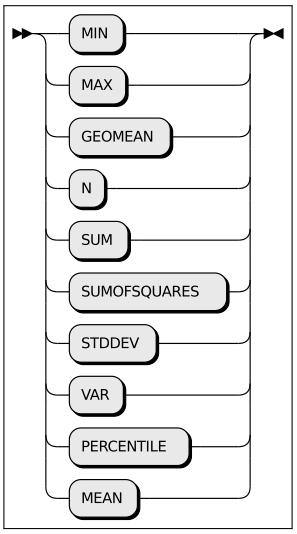


For the computation of aggregates based on performance metrics, the AggregateMetricClause provides necessary ways for expressing aggregate calculation. The calculation of aggregates is independent of the capabilities of a DQL Connector, thus implemented at the language level. The aggregate calculation is part of the DQL QEE and available aggregates are defined at the language level.

In order to calculate an aggregate, a user is able to choose a statistical operation as an instance of StatisticalAggregateType. The StatisticalAggregateType is used to select the operations for performing the calculation based on the available performance metrics. The available performance metrics are defined either as MetricStarClause or MetricClause. To provide additional information for the computation of aggregates, e.g. the percentile to find its corresponding value, the ConfigurationPropertiesClause can be used.

See Section 5.3.4, StatisticalAggregateType, MetricStarClause, MetricClause and ConfigurationPropertiesClause for more information.

3.5.5.2. Rule StatisticalAggregateType



To support users calculating aggregates based on performance metrics without any dependency on the underlying performance meta-model, DQL supports several statistical operations to calculate aggregates at the language level. The set of available statistical operations is based on Apache Commons Math[3], which is used to compute some of the available aggregates. The documentation of Apache Commons Math contains all necessary information on the available aggregate functions.

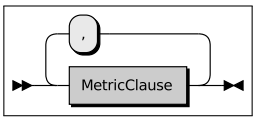
Remarks for Aggregate Functions

- `PERCENTILE`: This function requires the configuration property `percentile` with a value ranging from 1 to 100. The parameter defines which percentile should be computed.

See Section 5.3.4 and ConfigurationPropertiesClause for more information.

[3] `http://commons.apache.org/proper/commons-math/userguide/stat.html`

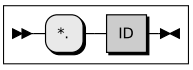
3.5.5.3. Rule MetricClauses



Through the rule MetricClauses, all metrics for computing an aggregate within a AggregateMetricClause can be referenced. Each instance of MetricClause contained in MetricClauses provides a single value for the aggregate computation process.

See AggregateMetricClause and MetricClause for more information.

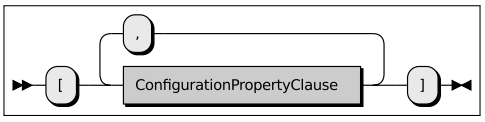
3.5.5.4. Rule MetricStarClause



To compute an aggregate for multiple model entities with the same metric name, the MetricStarClause is provided for convenience. A user can name the desired metric name through the ID non-terminal in this clause and each model entity found in the instance of the ForClause is added implicitly to the request for providing the named metric. The metrics will in turn be used to calculate the requested aggregate.

See ForClause for more information.

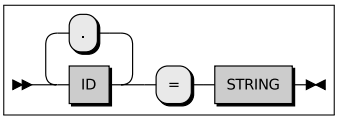
3.5.5.5. Rule ConfigurationPropertiesClause



For additional information that is required during the computation of aggregates, e.g. the value n to compute the n-th percentile for, the ConfigurationPropertiesClause allows to add meta-information for the computation process. The meta information is represented through one or multiple expressions of the type ConfigurationPropertyClause.

See StatisticalAggregateType and ConfigurationPropertyClause for more information.

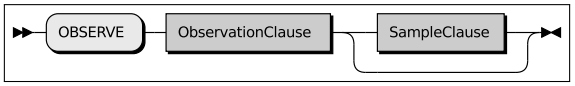
3.5.5.6. Rule ConfigurationPropertyClause



A ConfigurationPropertyClause allows to specify a key-value pair. A key consists of one or multiple parts of the non-terminal ID separated by the terminal `.`. The value in a ConfigurationPropertyClause is specified through the non-terminal STRING, which might be converted based on the key to provide the expected type. Key and value are separated by the terminal `=`.

3.5.6. Grammar Rules for Temporal Observations

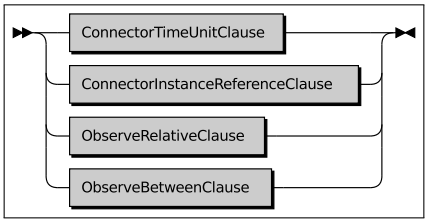
3.5.6.1. Rule ObserveClause



The ObserveClause allows to express a temporal dimension in DQL queries. By a temporal dimension, one or multiple different model instances might be used to compute performance metrics. The ObserveClause is initiated through the terminal `OBSERVE`, a specification of the observation formulated in the ObservationClause and a DQL Connector-specific SampleClause to specify samples over multiple model instances. The ObserveClause is orthogonally to Performance Metrics Queries as it does not change the query expression and the operations performed, but adds an additional dimension that the query needs to be solved for.

See Section 3.4.5, Section 5.3.5, Section 6.3.3, ObservationClause and SampleClause for more information.

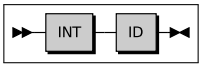
3.5.6.2. Rule ObservationClause



To specify an observation, the ObservationClause provides several options. A user might choose between a ConnectorTimeUnitClause or a ConnectorInstanceReferenceClause with DQL Connector-specific time units or reference marks and ObserveRelativeClause or ObserveBetweenClause to use time specifications defined by means of DQL.

See ConnectorTimeUnitClause, ConnectorInstanceReferenceClause, ObserveRelativeClause and ObserveBetweenClause for more information.

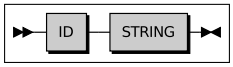
3.5.6.3. Rule ConnectorTimeUnitClause



In a ConnectorTimeUnitClause a user can express the amount of time units using the INT non-terminal and the identifier of the desired time unit through the ID non-terminal. The availability of time units and the interpretation of these units depends on the referenced DQL Connector.

See a DQL Connector-specific documentation, e.g. Section 6.3.3, for more information.

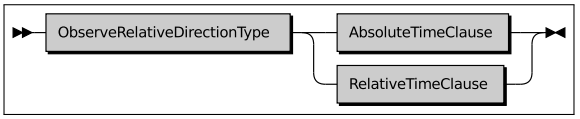
3.5.6.4. Rule ConnectorInstanceReferenceClause



In a ConnectorInstanceReferenceClause a user can express the kind of a reference through the ID non-terminal and the reference value through the STRING non-terminal. The availability of reference units and the interpretation of these units depends on the used DQL Connector.

See a DQL Connector-specific documentation, e.g. Section 6.3.4 for more information.

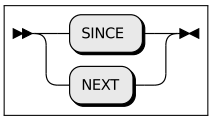
3.5.6.5. Rule ObserveRelativeClause



An ObserveRelativeClause consists of a direction in time specified through the ObserveRelativeDirectionType and either an AbsoluteTimeClause or a RelativeTimeClause to specify the bounds of the time window for observations defined through this clause.

See ObserveRelativeDirectionType, AbsoluteTimeClause and RelativeTimeClause for more information.

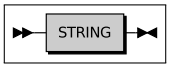
3.5.6.6. Rule ObserveRelativeDirectionType



To specify a relative time window, the ObserveRelativeDirectionType is used. By using the terminal `SINCE`, the time window is specified starting at a point in time in the past and spanned towards the present time. The terminal `NEXT` is the opposite, spanning a time window starting at the present time towards a future point in time.

See ObserveRelativeClause for more information.

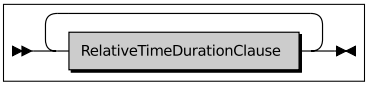
3.5.6.7. Rule AbsoluteTimeClause



An AbsoluteTimeClause specifies a formatted date string through the non-terminal STRING. The format is specified by the definition of date patterns of the class `java.text.SimpleDateFormat`[4] as part of the Java 6 Application Programming Interface (API).

See Section 5.3.5 for more information.

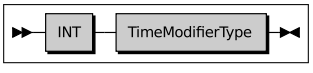
3.5.6.8. Rule RelativeTimeClause



A RelativeTimeClause consists of one or multiple instances of the type RelativeTimeDurationClause to specify an amount of time to span a time window using a reference point in time. The reference point is specified through other rules for time definitions.

See ObserveRelativeClause and ObserveBetweenClause for more information.

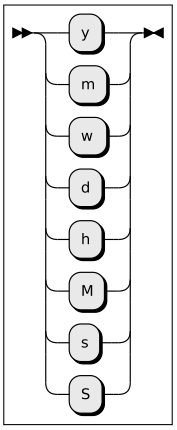
3.5.6.9. Rule RelativeTimeDurationClause



The definition of a time duration in an instance of RelativeTimeDurationClause is based on the non-terminal INT to specify the amount of time units and a TimeModifierType to specify the factor for calculating the time duration based on the time units.

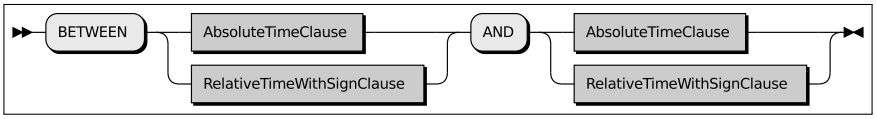
[4] `http://docs.oracle.com/javase/6/docs/api/java/text/SimpleDateFormat.html`

3.5.6.10. Rule TimeModifierType



The TimeModifierType contains terminals describing different time units. The time units and abbreviations as non-terminals are based on the class `java.text.SimpleDateFormat`[5] from the Java 6 API. As Millisecond is the smallest unit defined by this rule, time duration calculations are based on this resolution.

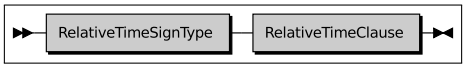
3.5.6.11. Rule ObserveBetweenClause



The ObserveBetweenClause is used to define a custom time frame. The terminal `BETWEEN` is the start of the rule followed by a time specification through either AbsoluteTimeClause or RelativeTimeWithSignClause as start of the time window, the terminal `AND` and another time specification to mark the end of the time window. As the calculation of the time window can be realized through various approaches, the actual calculation process is not defined at the language level, but at the implementation level of the DQL QEE.

See Section 5.3.5, AbsoluteTimeClause and RelativeTimeWithSignClause for more information.

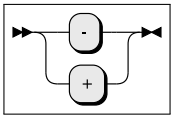
3.5.6.12. Rule RelativeTimeWithSignClause



A RelativeTimeWithSignClause extends the RelativeTimeClause by prefixing it with an instance of RelativeTimeSignType. The RelativeTimeSignType defines a relative time specification to be interpreted as directed into the past or the future.

See RelativeTimeSignType for more information.

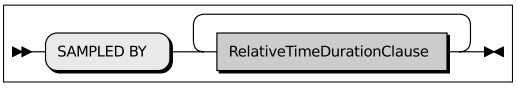
3.5.6.13. Rule RelativeTimeSignType



The RelativeTimeSignType expresses a time direction into the past through the terminal `-` or into the future through the terminal `+`.

[5] `http://docs.oracle.com/javase/6/docs/api/java/text/SimpleDateFormat.html`

3.5.6.14. Rule SampleClause



Through the SampleClause, a query resulting in multiple results, e.g. querying n model instances from the past through an `OBSERVE BETWEEN` expression or an `OBSERVE SINCE` expression, can be aggregated to samples. The size of a sample can be defined through using one or multiple instances of the rule RelativeTimeDurationClause to define the duration, i.e. the size, of a single sample. The rule is initiated through the terminal `SAMPLED BY`.

The availability of this sampling mechanism depends on the referenced DQL Connector and not defined through by means of DQL.

3.5.7. Extensions for Constrained Queries

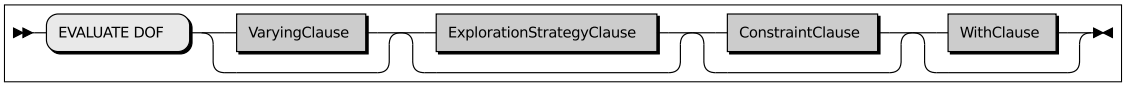
3.5.7.1. Rule ConstraintClause

The ConstraintClause is used to provide an advice for solving the model instance and to calculate performance metrics. It is an extension of the SelectQuery and used to build queries from the class of Constrained Queries.

See Section 3.3.4, Section 3.4.3 and SelectQuery for more information.

3.5.8. Extensions for Degree-of-Freedom Queries

3.5.8.1. Rule DoFClause

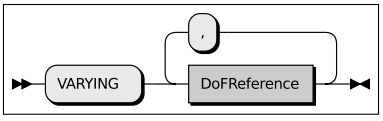


The DoFClause extends a SelectQuery to analyze performance model instances under varying parametric settings, i.e. DoFs. The rule consists of four additional rules that can be added optionally to control the exploration of the DoFs. The non-terminal initiating the rule is `EVALUATE DOF`.

Using a VaryingClause the relevant DoFs can be selected explicitly, using a ExplorationStrategyClause the strategy for the exploration of DoF configuration spaces can be set, the ConstraintClause allows to define the trade-off between accuracy and response time and the WithClause allows to specify an additional solver for DoFs besides of the DQL Connector.

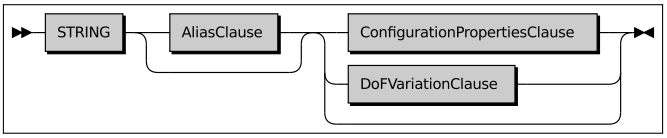
See Section 3.3.4, Section 3.4.4, SelectQuery, VaryingClause, ExplorationStrategyClause, ConstraintClause and WithClause for more information.

3.5.8.2. Rule VaryingClause



The VaryingClause is initialized through the non-terminal `VARYING` and allows to specify one or multiple instances of non-terminal rules of the type DoFReference. This rule allows users to specify which DoFs should be varied during the evaluation of DoFs.

3.5.8.3. Rule DoFReference

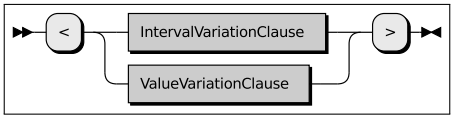


A DoFReference is a specialized variant of the EntityReference for referencing DoF entities in a performance model instance. By means of DQL, a DoF is an entity in a model instance that can be used for varying its parametric settings in order to analyze the performance dynamics of a model instance. The exact definition of model entities serving as DoF is subject to the referenced DQL Connector.

The DoFReference consists of a non-terminal of the type STRING as the identifier in the model instance of the DoF, an optional alias definition through an instance of AliasClause and optionally either a ConfigurationPropertiesClause or a DoFVariationClause, which is a specialized ConfigurationPropertiesClause.

See AliasClause, ConfigurationPropertiesClause and DoFVariationClause for more information.

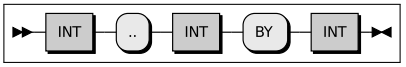
3.5.8.4. Rule DoFVariationClause



A DoFVariationClause specializes the behavior of a ConfigurationPropertiesClause for expressing the variation of a DoF shorthand. The rule is initiated by the terminal < followed by either an IntervalVariationClause or a ValueVariationClause to express the valid DoF parameter variation space and is terminated by the terminal >.

See Section 5.3.6, IntervalVariationClause, ValueVariationClause and ConfigurationPropertiesClause for more information.

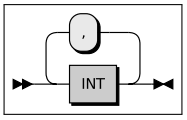
3.5.8.5. Rule IntervalVariationClause



Using an IntervalVariationClause, the range of valid parameter values can be specified. The parameter space is evaluated through an initial value, i.e. the first non-terminal INT, a bound specification, i.e. the second non-terminal INT, and an increment, i.e. the third non-terminal INT. The final semantics for evaluating the clause are subject to the implementation.

See Section 5.3.6 for more information.

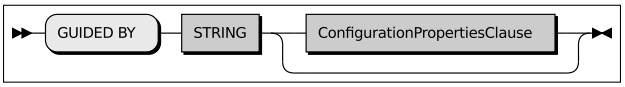
3.5.8.6. Rule ValueVariationClause



A ValueVariationClause is used to specify an absolute list of possible parameter settings for a DoF. The rule consists of one or multiple non-terminal INT instances separated by a terminal ,.

See Section 5.3.6 for more information.

3.5.8.7. Rule ExplorationStrategyClause



If a DQL Connector supports different exploration strategies, an ExplorationStrategyClause can be used to specify which exploration strategy is to be used. The exploration strategy is specified through an instance of the non-terminal STRING rule and can be, if necessary, configured through a ConfigurationPropertiesClause.

See Section 5.3.6 and ConfigurationPropertiesClause for more information.

3.5.9. Grammar Rules for Performance Issue Queries

3.5.9.1. Rule PerformanceIssueQuery

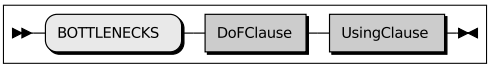
In PerformanceIssueQuery the top-level queries for the class of Performance Issue Queries are aggregated. Currently the class is solely based on the rule DetectQuery.

3.5.9.2. Rule DetectQuery

A DetectQuery is used to start an automated analysis of a performance model instance for a specific cause. The rule is initiated through the terminal `DETECT` and followed by the non-terminal DetectBottlenecksQuery. It is used to express which automated analysis shall be conducted.

See DetectBottlenecksQuery for more information.

3.5.9.3. Rule DetectBottlenecksQuery



The DetectBottlenecksQuery is initiated by the terminal `BOTTLENECKS` followed by a non-terminal DoFClause and a non-terminal UsingClause. Using this rule, a DQL Connector can be queried to analyze a performance model instance for bottlenecks limiting the overall system performance. The exploration of DoFs helps to determine the significance of parameter influences.

See Section 4.4 for more information.

4. Related Work

Supplementary to Chapter 3, this chapter presents related approaches to reason on the design and foundations of DQL. In Section 4.1 intermediate modeling approaches will be outlined to be used as assisting technique in SPE for analyzing performance model instances. Section 4.2 focuses on formal approaches being used to capture dynamics in performance model instances and Section 4.4 is related to a specialized analysis technique answering also qualitative questions through analyzing performance model instances. With a special emphasis on the textual syntax of DQL, Section 4.5 will complete the related work with an outlook to Structured Query Language (SQL) as a successful approach in the field of DSLs.

4.1. Intermediate Models in Performance Engineering

Intermediate modeling approaches allow to generalize transformation processes through an intermediate step. The intermediate step is based on a predefined intermediate model, that has a specified set of transformation rules either for incoming, outgoing or both directions. Once a set of transformation rules has been defined, the rules can be reused for other model instances if the rule set is based on the meta-model level. Being more general, intermediate models cope with the problem having of n input formats and m output formats (*N-to-M problem*). In the SPE domain such approaches can be used for the analysis of performance models as follows.

In [SL11] an overview of the history and development of approaches for model interoperability in the SPE domain is presented. Starting at interchange formats for tool interoperability, more recent intermediate model approaches are introduced. The foundation of the most recent approaches is based on Performance Model Interchange Format (PMIF) in its enhanced version S-PMIF [SLP10]. S-PMIF allowed users to cover the complete process from design models to analysis models with the adequate tooling for each step.

Approaches specialized on intermediate models are Performance by Unified Model Analysis (PUMA) with Core Scenario Model (CSM) and Kernel LAnguage for PErformance and Reliability analysis (KLAPER), both addressing the N-to-M problem. Both approaches have their foundation in different software design domains [GMS06, WPP+05]. The PUMA approach is based on the challenge to analyze design models in UML Profile for Schedulability, Performance and Time (UML-SPT) based on UML [Obj05]. The intermediate step in PUMA refers to CSM, which is then used to transform into an analysis model, e.g.

into a Petri Net (PN), QN or Layered Queueing Network (LQN), for simulation purposes. Hidden in the CSM step is the transformation and parameterization of the analysis model to unburden users from manual work.

KLAPER differs significantly from PUMA in the following points: (i) KLAPER is focused on CBSE, (ii) makes no specific assumption about the incoming design model and (iii) allows the individual parameterization of the analysis model in the intermediate step through its own Extensible Markup Language (XML)-based language [GMS06, GMRS08, Koz10]. Thus KLAPER can be seen as being more general, but at the cost of effort for defining initial incoming transformations and the parameterization if not using an automated approach.

Through intermediate models automation approaches for the analyses of performance models have been developed. The approaches, once all transformations rules have been implemented, see [GMRS08, p. 329 ff.] for remarks on the effort, allow the automated analysis from the design model to the analysis model. Therefore users of these approaches are unburdened from the manual transformation of performance model instances, but they are forced to decide which model solving techniques shall be used for the analysis.

Besides of the decision of choosing the right model solving technique, the intermediate approach is focused on offline execution and offers no support for requirements in an online scenario, e.g. a trade-off between accuracy and speed of predictions.

4.2. Modeling of Degrees-of-Freedom and Strategies for their Exploration

As modern computing environments become more dynamic, e.g. through the reconfiguration of run-time aspects like available vCPUs, architecture-level performance models need to offer facilities for modeling these aspects. As example we refer to the Adaptation Points Meta-Model as part of DMM [BHK12, HBK12]. In this example, the Adaptation Points meta-model can be used to enrich an existing resource landscape model instance to express certain DoFs of deployment resources. For instance a VM could be annotated as being configurable to operate on 1 to n vCPUs. Further details on this topic can be found in Section 2.2.

Besides of the modeling aspect, from DoFs new challenges are opened. For performance predictions and their derived reconfiguration processes in real-world scenarios, the reconfiguration process needs to be modeled. This model reflects the valid configuration space of the DoFs and the conforming reconfiguration space in the real-world environment. Finding an optimal solution for the reconfiguration then is in essence an optimization problem, where a solution, e.g. a Pareto-optimal solution, needs to be found [KR11].

An important aspect of DoFs and the number of configuration options (d_i) for a given DoF, is the rapidly growing configuration space. DoFs are depending on each other, thus leading to a total number of $D = \prod_i^n d_i$ valid model instance configuration settings representing the experiment runs to be conducted. The configuration space might be growing even further when considering more complex parametric dependencies, e.g. due to constraints arising from hardware configurations or software licensing. Thus the simulation of all possible configuration options for finding an at least Pareto-optimal solution, causes high computing demand that might exceed available capacities.

To cope with this challenge, advanced methods for the exploration of the configuration space are needed. A viable solution for this problem is the usage of statistical models for the configuration space and a sensitivity analysis for finding satisfying results with reasonable simulation effort. In [WKH11] the concept of software performance curves is

proposed. The concept is solely based on statistical reasoning about model parameters without the need of any knowledge of the underlying system.

When applying software performance curves, an efficient experiment selection can reduce the total number of experimental setups to analyze. Due to the statistical model, results for missing simulation points (i.e. combinations of configuration options) can be interpolated and the next experiments can be selected based on statistical relevance. Concluding in [WHKF12] statistical methods for parameter selection are evaluated for their prediction quality. The results show that a reduction of the number of experiments with respect to statistical errors is possible and a viable option for further research activities.

4.3. Approaches for the Modeling of Performance Metrics and Measurement

This section will introduce modeling techniques for the modeling of measures. Using standardized meta-models for the modeling of measures, the results of tools with different model solving approaches become comparable and users gain more flexibility. As one approach for the modeling of metrics, the OMG introduced the Structured Metrics Metamodel (SMM) as part of their Architecture Driven Modernization (ADM) roadmap [Obj12].

SMM is focused on the use in scenarios related to software technology and contains exemplary metrics for the Software Engineering (SE) domain, but is not limited to this domain. Using SMM, any kind of structured metric can be modeled, measured and represented. When using SMM for measuring, users can follow a process similar to (i) defining measures by new SMM instances, (ii) apply the measures to source model instances of a software system and (iii) retrieve measurement results.

Consistently with previous works of OMG, SMM is defined as instance of the MOF Meta-Model. This allows integration with other meta-models of the OMG, e.g. with UML [Obj11a, Obj12]. Concluding the intention of SMM can be formulated as: *"This executable measuring should enable another tool vendor, presented with the same measure libraries, observation information and instance models, to be able to apply those measures in an unambiguous fashion and to come up with the same measurements (subject to uncertainty errors)"* [Obj12].

As an example for the implementation of SMM, Measurement Architecture for Model-Based Analysis (MAMBA) supports a wide-range of SMM features and adds additional features. One notable addition of MAMBA is MAMBA Query Language (MQL) as interface to metrics. MAMBA is an implementation of SMM based on Ecore instead of MOF as underlying meta-model. Opposite to SMM, MAMBA offers a variety of tools for the execution of queries against model instances [FvHJ+11, FJKH12].

MQL has a SQL-like textual syntax and supports, advanced of the core functionality of SMM, aggregates on measurements. In addition also a run-time environment for MAMBA exists, that allows continuous queries of model instances. An exemplary query of MQL can be seen in the following:

```
1 select AverageMethodResponseTime("exampleMethod") as avgrt
2 from myApp where avgrt > 500 group by kdm.code.MethodUnit
```

Listing 4.1: MQL Example from [FJKH12]

The structure of the query example shows a direct relation to the structure of SQL. SQL is a common example for a DSL as a query language and it allows to hide the actual data access and calculation from the user.

4.4. Detection of Bottlenecks in Performance Models

In [WFP07] the authors outline the identification of bottlenecks as a direction for SPE research. A bottleneck is defined as a resource, limiting the performance of a whole system due to saturation. As already described by [DB78] for a given workload profile, each kind of queuing network has at least one bottleneck. The analysis of bottlenecks and especially the judgment on their impact can be seen as integral part of SPE besides of predicting raw performance metrics, e.g. predicting the exact queue length of a resource serving requests.

Due to the increasing complexity of software systems and their sophisticated resource allocations, e.g. message-driven software systems with exclusive resource pools for different message types and different QoS classes, searching for bottlenecks is crucial for efficient operation. The dynamics of these systems can be described and approximated by layered bottlenecks with layered resources and services as in [FPW+06].

In these models, the execution of a service depends on other services to be executed, implying a hierarchical resource possession. These hierarchies imply that a single bottleneck might be caused by a composition of multiple other bottlenecks. This leads to the demand for specialized techniques of bottleneck detection, besides of judging between the states *saturated* or *not-saturated*.

4.5. Domain-specific Languages for Modeling Queries

A DSL is a kind of a programming language or set of formal expressions to express a specific problem. Often a DSL has only a limited expressiveness to formulate problems. The reduced expressiveness is advantageous in order to reduce the effort for learning a language, but on the other side the language elements of a DSL composed of general instructions might not be as efficient for computation as implemented manually. Leveraging the concept of DSLs, users can increase their productivity through the use of a DSL in certain scenarios but for the compromise to be able to use the language just in a single domain-context [Fow10].

One widespread approach of a DSL for accessing structured data is SQL. It is based on relations as structure for organizing data and can be used to retrieve and manipulate data or alter the structure of relations. The roots of SQL are in Structured English Query Language (SEQUEL), which is intended to offer users of different experience levels a unified interface to relational databases [CB76].

In SEQUEL users can formulate statements in a declarative way. The underlying Relational Database Management System (RDBMS) has to break down queries and translate them into instructions that can be executed on the underlying computing environment. In this way users are unburdened from platform characteristics like file access, object storage locations and more complex tasks like transaction management or physical data layouts.

Furthermore, besides of human users, the strict structure of statements in SQL allows also machine users to use SQL as an API to RDBMS. An example of SQL as an interface for machine users is Java Persistence API (JPA). When using JPA all SQL is generated hidden from the software developer and executed without any human interaction [DeM09]. From a software developer's view, the developer is only interacting with Enterprise JavaBeans (EJBs) and the execution container (i.e. an Java Application Server with JPA) is managing the direct interaction with the underlying RDBMS [Sak09].

The approach of SQL shows that a DSL with a declarative and strictly structured formalism is appropriate for both human and machine users. Moreover this kind of interface unburdens users from detailed knowledge about the underlying system characteristics and environment involved in executing queries. Hence, concepts of SEQUEL and SQL are a viable foundation for the design of a DSL for online performance queries.

5. Implementation of a Query Language for Online Performance Queries

This chapter explains the approach of implementing DQL. It uses the usage scenario descriptions and the resulting textual syntax described in Chapter 3 to compose an architecture capable of the requirements and to provide an implementation suitable as a first starting point for later development. At first the system architecture consisting of several components will be introduced in Section 5.1. As an approach of an abstraction layer to support different performance modeling approaches, the implementation is built around the Mapping Meta-Model. The usage of the Mapping Meta-Model and the role as an abstraction layer is explained in Section 5.2. The interaction of components and conceptual realization of processing queries is presented in Section 5.3. The chapter is finished by presenting the UI realized to provide users an Eclipse-based interface in Section 5.4

5.1. System Architecture and Component Description

5.1.1. Description of the System Architecture

5.1.1.1. Introduction of the System Architecture

The implementation of DQL is based on the principles of CBSE with a separation of concerns between the different implemented software components. The foundation for the encapsulation and interaction of components is realized through OSGi Bundles and the OSGi Service Layer as introduced in Section 2.4. The minimal set of components to assemble a complete run-time environment for DQL can be seen in Figure 5.1. These components can be deployed on OSGi-compliant run-time environments, e.g. Eclipse Equinox, which is the designated development platform.

The components shown in Figure 5.1 are separated in order to allow the flexible development of new features. Each component is focused on a single concern and provides interfaces to their consumers in order to use the functionality provided by the component. Furthermore, as CBSE suggests, the separation allows to exchange component implementations, while reducing the impact of changes. To change the implementation of a software component, the changes are hidden by interfaces, which allows to develop parts of DQL independently. This section will present a brief overview of the components, while Section 5.1.2 will present detailed descriptions of the interfaces provided by components.

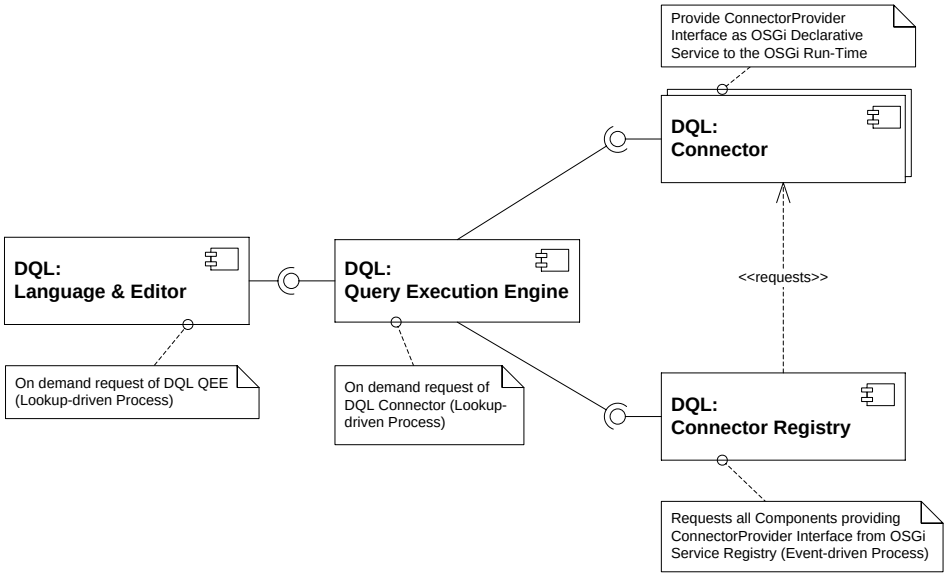


Figure 5.1.: Overview of Components and Dependencies

5.1.1.2. DQL Language & Editor Component

The DQL Language & Editor Component is the result of the code generation facility provided by Xtext, see Section 2.3, and the grammar of DQL as input. The component consists of several sub-components to provide the parser and lexer for DQL statements, an Eclipse EMF-based model to represent a DQL statement and an editor for the Eclipse IDE with user assistance features, e.g. syntax highlighting and content assist, to write DQL queries.

This component can be assembled to fit demands. In a batch usage scenario, where the run-time for DQL might be deployed on a headless server, the sub-components for the editor should be removed from the assembly. Whereas in a desktop deployment, the sub-components providing the editor would provide a valuable user interface. For the consuming component, the DQL QEE, the EMF representation of the query is the relevant result of this component to process the query further.

5.1.1.3. DQL Query Execution Engine (QEE)

The DQL QEE is the central hub for executing queries. Based on the raw EMF representation of the DQL query provided by the DQL Language & Editor Component, the QEE interprets the query and performs semantic checks of the requested computation. Semantic checks are applied, f.i. in the MetricReferenceClauses for entities not referenced through the ForClause, and the query processing is aborted if the QEE detects any severe errors.

The QEE is also responsible for the calculation of aggregates, see Section 5.3.4, and the interpretation of the temporal dimension of queries, see Section 5.3.5. Additional time units may be provided through functionality of the DQL Connector implementation, in Section 6.3 an example is presented. To delegate the computation of performance metrics and accessing performance model instances, the QEE utilizes DQL Connectors that are being looked up through querying a centralized DQL Connector Registry.

Through the strict decoupling of parsing, interpreting and executing queries, taking place at the QEE layer, and the solving of model-centric queries, taking place at the Connector

layer, the isolated QEE component serves as central abstraction layer. The centralization of abstraction allows either replacing the current DQL implementation by a new approach or to replace the solver encapsulation approach through a more sophisticated subsystem than the current approach of the Connector layer. Thus the architecture enables future improvements through replacing parts of the application without direct impact on other layers.

5.1.1.4. DQL Connector

A DQL Connector is a OSGi Bundle for interpreting queries of a specific performance meta-model family, e.g. one Connector might be implemented for DMM and another Connector might be implemented for Palladio Component Model (PCM). Thus the term *Connector* is a synonym for the realization of a performance meta-model plug-in, but as each OSGi Bundle might be considered as plug-in, the term Connector shall avoid confusion and emphasize the relation to the DQL context. In a query, a specific Connector is named through the ModelFamily in the UsingClause provided by the user.

The implementation of a Connector has to provide a set of Java Interfaces that is introduced in Section 5.1.2 and its availability is announced through the OSGi mechanism for *Declarative Services*. The Declarative Service is exposed together with the implementation of a Java Interface and registered as a service Section 2.4 provides additional information about the OSGi Service Layer.

In order to realize the computation of performance metrics for requests, the implementer of the Connector is free to choose how to implement the Connector. As the performance meta-model families differ significantly, e.g. comparing DMM to a meta-model for Queueing Petri Networks (QPNs), the implementer might only implement parts of the set of available classes and their functionality for Performance Metric Queries. Due to the CBSE-based implementation approach, only the relevant interfaces need to be provided by the deployed component while others are omitted.

5.1.1.5. DQL Connector Registry

As DQL should support multiple performance meta-model families, the DQL Connector Registry performs the bookkeeping of available DQL Connectors for the different performance meta-model families. The need for a Connector Registry arises from the fact, that OSGi has no explicit support for different OSGi Bundles providing the same functionality, i.e. implementing the same Java Interfaces, and to select one of these Bundles by an additional property. OSGi offers only a ranking mechanism for providing a service by declaring a rank [OSG11, p. 116]. A detailed description of the registration mechanism realized for DQL using Declarative Services will follow in Section 5.1.2.2.

The registration process follows the OSGi Lifecycle, i.e. triggered through the lifecycle of DQL Connector Bundles resulting in an event-oriented process. After completion, the DQL QEE can request references to DQL Connectors, which can be used to create an instance of a DQL Connector for a specific performance meta-model family and the desired query class.

Thus using the Connector Registry, the effort for looking up DQL Connectors has a small footprint requiring only to access two instances of the associative data structure `java.util.HashMap` and is aligned to the OSGi run-time. Without the Connector Registry, the look up of the DQL QEE would result in searching each available Bundle for the requested Java Interface and supported performance meta-model family.

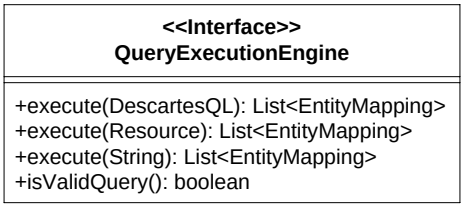


Figure 5.2.: Interfaces provided by the Query Execution Engine

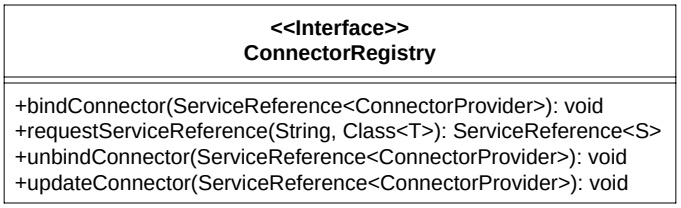


Figure 5.3.: Interfaces provided by the Connector Registry

5.1.2. Description of Interfaces within the System Architecture

5.1.2.1. Interfaces provided by the Query Execution Engine

The DQL QEE is the starting point for executing DQL queries. A query can be submitted to the QEE for computing performance metrics or other information. The relevant methods provided by the QEE are shown in Figure 5.2. To submit a query, the QEE offers the `execute`-methods with support for different input formats. A user can decide to submit a query as instance of `DescartesQL`, which is an EMF-based model instance provided by the Xtext parser, as instance of `Resource`, which is a EMF Resource to load the Xtext model instance from persistent storage, or as instance of `String`, which is a raw textual representation of the query.

Internally in the current implementation state, the `execute`-methods reuse the `execute(DescartesQL)` method to provide results. The `execute`-methods are blocking. Results of the execution are returned as `List<EntityMapping>`. If the results should be treated as valid by the caller, i.e. during the whole execution process no error or exception happened, the `isValidQuery` method returns `true`. Otherwise the results should not be used for further processing. The problem needs to be analyzed manually in this case, e.g. through analyzing logs of QEE or the referenced Connector.

5.1.2.2. Interfaces provided by the Connector Registry

The DQL Connector Registry provides an unified interface for interaction with the OSGi Service Layer and the DQL QEE to request a reference to create an instance of a specific DQL Connector. Figure 5.3 shows the relevant methods of the unified interface.

For the interaction with the OSGi Service Layer, the methods `bindConnector`, `unbindConnector` and `updateConnector` are necessary. They are part of the lifecycle management to maintain the references to available DQL Connectors. These methods are used only internally and the mapping of the OSGi Lifecycle to means of DQL will be explained in Section 5.3.1.

The method `requestServiceReference` is used to retrieve a reference to a `ConnectorProvider`, see Section 5.1.2.3, serving for a specific performance meta-model family. The performance meta-model family is specified as `String` and a specific query class is requested as second parameter of the type `Class<T>`. The separation of support for a

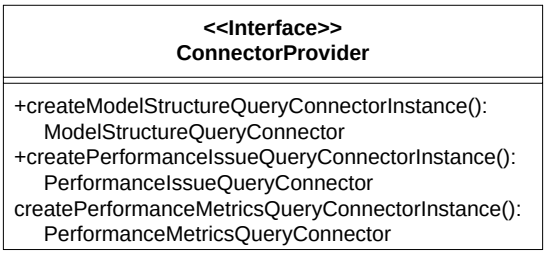


Figure 5.4.: Interfaces provided by a Connector Implementation to the Connector Registry

specific performance meta-model family and support of a specific query class allows to span the support of a specific performance meta-model family across multiple Connectors.

5.1.2.3. Interfaces provided by a Connector Implementation

The DQL Connector provides two interfaces for executing queries on the performance model instance. In Figure 5.4 the `ConnectorProvider` interface is shown. The implementation of this interface is instantiated through an OSGi `ServiceReference` obtained from the DQL Connector Registry. After an instance of `ConnectorProvider` has been created, an instance of a type derived from `QueryConnector`, as shown in Figure 5.5, can be created. The interface is based on the factory approach.

A `ConnectorProvider` implementation allows to create instances of Connectors for specific query classes, introduced in Section 3.2, realized as implementations of the interfaces `ModelStructureQueryConnector`, `PerformanceMetricsQueryConnector` or `PerformanceIssueQueryConnector`. A Connector implementation might therefore only implement the interfaces that are actually available for the performance meta-model family and is not forced to provide stubs.

Furthermore for each `request`-method shown in Figure 5.5, a complementary `supports`-method exists indicating whether a request in a query class is supported. In case of the class of Performance Metrics Queries, support for basic performance metrics through the `requestMetrics`-method might be available, but support for analyzing DoFs through the `requestDoFMetrics`-method might be unavailable. Testing for supported methods allows a transparent exception handling and error reporting to users while using Connectors from arbitrary sources. The referenced type `EntityMapping`, used as data structure for requests and responses between the DQL QEE and the implementation of Connectors, will be introduced in Section 5.2.

For the interfaces `PerformanceMetricsQueryConnector` and `PerformanceIssueQueryConnector`, which are both being derived from the `StatefulQueryConnector` interface, an additional `reset`-method exists. As `PerformanceMetricsQueryConnector` and `PerformanceIssueQueryConnector` allow to configure their internal behavior through additional `get`- and `set`-methods[1], this method is used to reset the Connector instance back to an reentrant state.

An example for a configurable behavior is the selected exploration strategy for the `requestDoFMetrics`-method in the `PerformanceMetricsQueryConnector`. As the Connector implementer might reuse cached objects within a Connector component, the `reset`-method is used to indicate a new request that shall be interpreted using the default values and reset any Connector-internal state, that might have misleading impact on the results.

[1] Omitted in Figure 5.5 to support the overview.

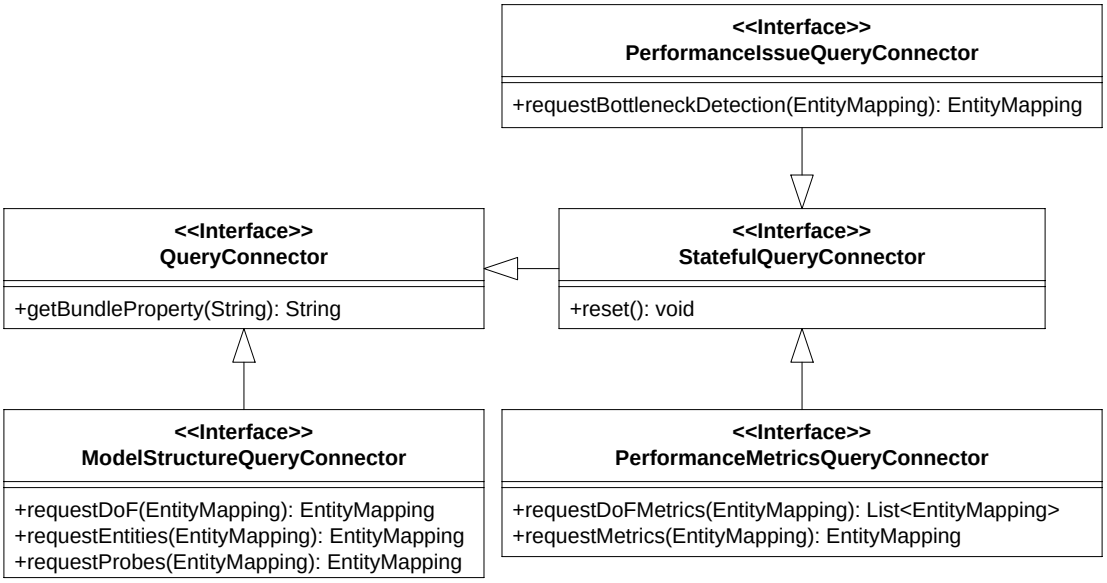


Figure 5.5.: Interfaces provided by a Connector Implementation for Query Execution

Thus the `reset`-method ensures a reentrant Connector implementation, while allowing Connector-internal optimization.

5.2. The Mapping Meta-Model

5.2.1. Introduction of the Mapping Meta-Model

The Mapping Meta-Model is an EMF-based meta-model serving as an abstraction layer between DQL queries and specific performance meta-model families. Furthermore it is used to represent the results of queries and can be used as a persistence mechanism.

The focus of the Mapping Meta-Model is to (i) retain information of a architecture-level performance model instance in a well-structured and generalized fashion and (ii) to represent DQL query contents in an abstract way with focus on the structural properties of the queried performance model instance. The term structural properties is in this case defined to be centric to the model entities of performance model instances together with their computable performance metrics. These structural properties can be requested through instances of the Mapping Meta-Model, while the relationship, i.e. the architectural details, among the entities is hidden.

Thus, due to missing relationships among model entities, the Mapping Meta-Model is not intended to replace neither architecture-level modeling approaches, nor performance modeling approaches. Detailed descriptions of the meta-model, the usage as abstraction layer between the DQL QEE and the DQL Connectors will follow in the subsequent sections. In Section 6.3 an approach for extending the Mapping Meta-Model into a Performance Data Repository (PDR) for persisting query results and to provide access to historic results will follow.

5.2.2. Description of Model Entities

In Figure 5.6 the Mapping Meta-Model is shown. The meta-model is made up on three layers forming a generic representation of structural properties extracted from a performance model instance.

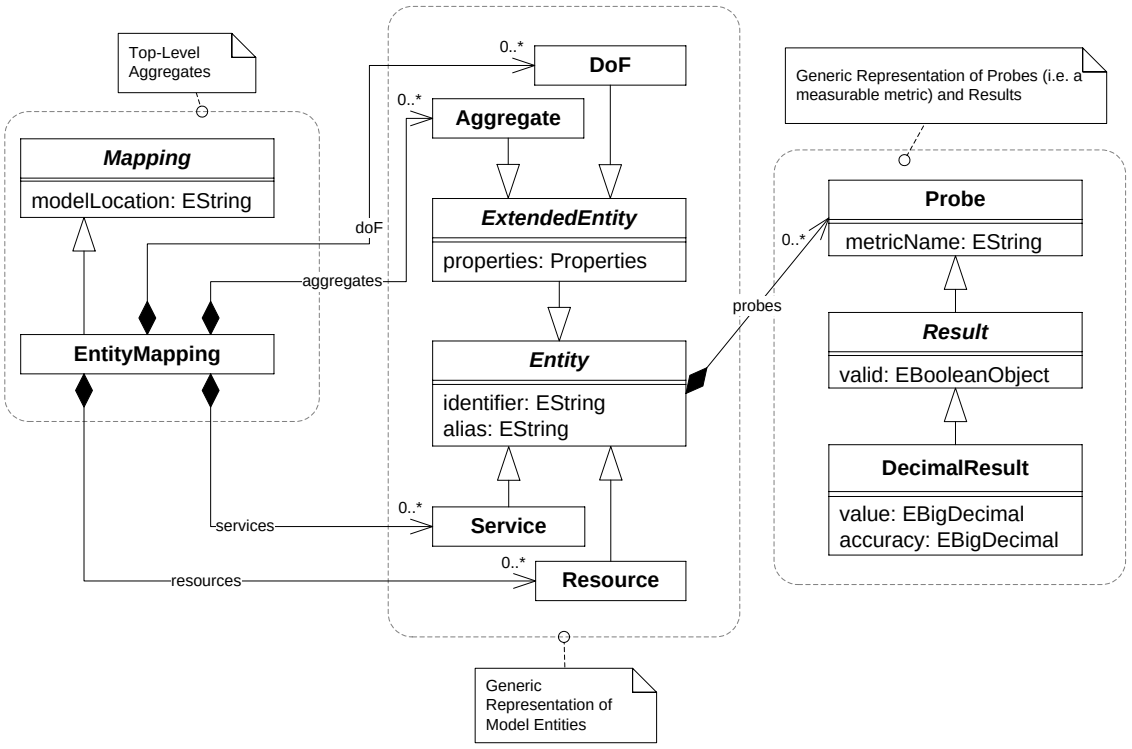


Figure 5.6.: Diagram of the Mapping Meta-Model

The *Top-Level Aggregates* are on the left side in Figure 5.6. `EntityMapping`, derived from the abstract type `Mapping`, is used to aggregate all information retrieved from a performance model instance by using computations or to submit a request for performance metrics. The property `modelLocation` stores an URI interpretable by the utilized DQL Connector. The containers for `aggregates`, `doF`, `resources` and `services` store references to representations of different model entities found in architecture-level performance model instances.

In `resources` and `services` the representations of performance-relevant entities are stored. The referenced types are in the center of Figure 5.6 and part of the layer for *Generic Representation of Model Entities*. Both are derived from the abstract type `Entity` with the properties `identifier` to store an absolute identifier of the model entity as used in the performance model instance and `alias` to store any kind of identifier the user suggested in the originating DQL query, see AliasClause. To request and store performance metrics, the `Entity` type allows to store references to the `Probe` type, which will be described in the subsequent paragraphs.

The remaining attributes of `EntityMapping` of the types `Aggregate` and `DoF` are derived from the type `ExtendedEntity` with an additional attribute `properties` of the type `java.util.Properties` from the Java API. The `properties` attribute is intended to store meta-information that is supplementary to the `ExtendedEntity` and required to interpret the instance referenced. In case of the type `Aggregate`, which is used to store results of the DQL QEE for computed aggregates, see Section 5.3.4, these information are related to the computation process of an aggregate. For the `DoF` type, which is used to represent a DoF and its setting, `properties` may contain information on the DoF variation. The value of the `Aggregate`, i.e. the result, and `DoF`, i.e. the decimal value of a DoF in the current variation, is stored through the last part of the meta-model, the layer for *Generic*

Representation of Probes and Results on the right.

The `Probe` type is used to specify the computation of a performance metric, that can be identified through the attribute `metricName`. As identifiers of metrics might vary between different performance meta-model families, the `metricName` is supplied as `String`. When the result of a computation is returned through an instance of the Mapping Meta-Model, `Probes` are replaced through sub-types of the abstract type `Result`. A `Result` has the property `valid` to indicate whether the necessary computation processes have finished without any errors and if the result value is eligible to be used for analysis by the user. Currently, the Mapping Meta-Model only offers the type `DecimalResult` as result value. Other types may be added in future revisions. The `DecimalResult` allows to store a `value` of the result and its `accuracy` or the significance for statistical values, both represented as the type `Double`.

To illustrate the usage of the Mapping Meta-Model, Section 5.2.3 and Section 5.2.4 provide examples and exemplary instances of the meta-model. The usage of the attributes `aggregates` and `doF` in the `EntityMapping` type will be explained in the sections Section 5.3.4 and Section 5.3.6.

5.2.3. Usage in Model Structure Queries

The Mapping Meta-Model supports Model Structure Queries in order to provide an overview of queryable entities for a given performance model instance. This section describes the usage of the Mapping Meta-Model between the DQL QEE and DQL Connectors. For queries from this class, the QEE uses the response of the Connector to pass it back to the user as response.

In case of a query using the `LIST ENTITIES` expression, the result provided by the DQL Connector contains all resources and services recognized in the performance model instance. For this query type, the Mapping Meta-Model is used to form the request and the response. Figure 5.7 shows exemplary instances for the request and response pair used. The request on the left contains an instance of the `EntityMapping` with the `modelLocation` attribute set. The DQL Connector can therefore look up the performance model instance and query it through its internal operations. The response on the right shows an instance of `Service` and an instance of `Resource`, which could be found in the performance model instance. For both instances, the `identifier` is set, but the `alias` is empty as it is not yet set by the Connector. A DQL Connector may suggest an alias.

Following the exemplary workflow for performance analysis using DQL as suggested in Section 3.4, the subsequent query based on the `LIST METRICS` expression is intended to reveal the available performance metrics for a set of explicitly specified entities. Figure 5.8 shows the corresponding request and response between DQL QEE and the DQL Connector. The request contains an instance of `EntityMapping` and names the relevant instances of `Resource` and `Service` with their `identifier` attribute set properly.

In the response on the right of Figure 5.8, the original request is extended and contains instances of `Probe` with the `metricName` attribute set. These `Probes` indicate computable metrics and are individual for their referencing entity, i.e. *metric1* is available for the resource aliased as *res1*, but not for the service with the alias *svc1*.

5.2.4. Usage in Performance Metrics Queries

In Figure 5.9 the example of the Mapping Meta-Model for a `SELECT` expression is shown. The request is formed through submitting an instance of the `EntityMapping` and naming all relevant entities as instances of the types `Resource` or `Service` and the desired metrics as `Probe`. In this example, the request is similar to the response shown in Figure 5.8.

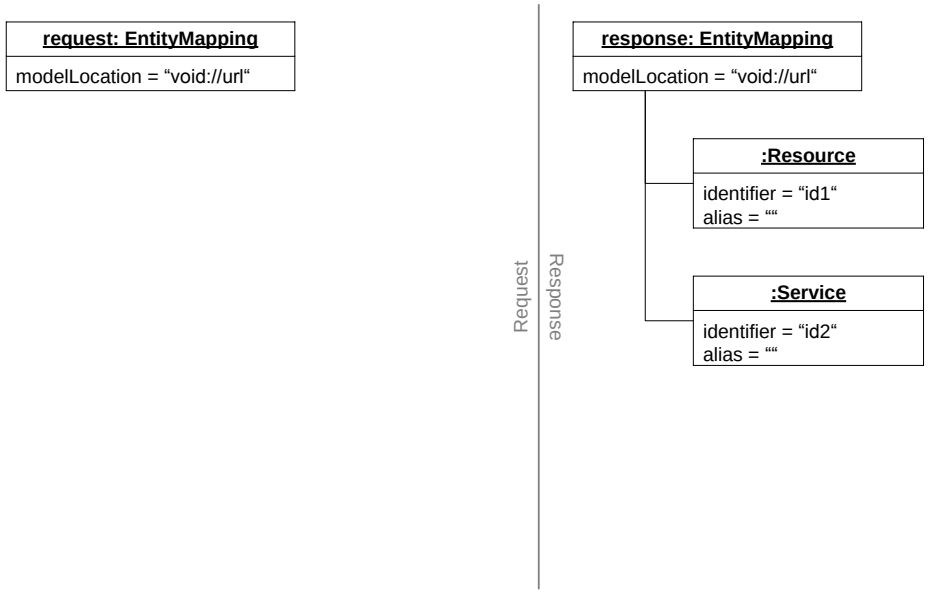


Figure 5.7.: Instances of the Mapping Meta-Model representing the Query from Listing 3.1

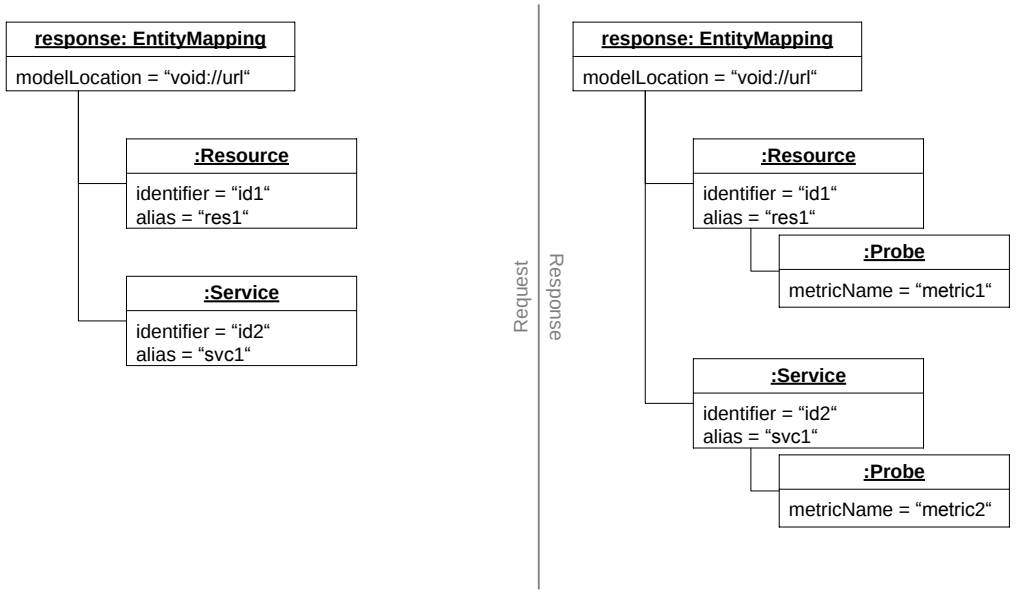


Figure 5.8.: Instances of the Mapping Meta-Model representing the Query from Listing 3.2

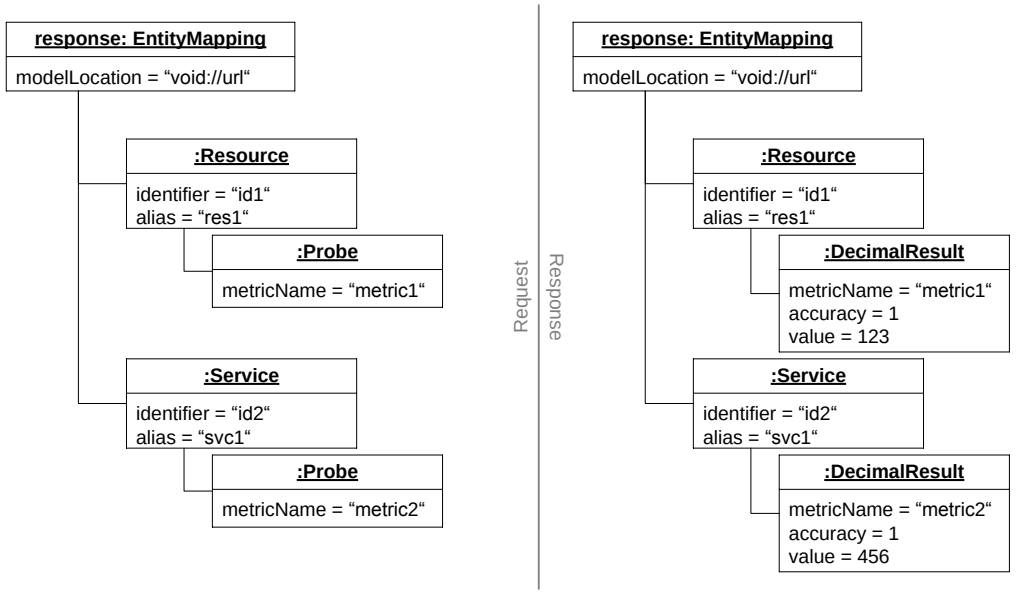


Figure 5.9.: Instances of the Mapping Meta-Model representing the Query from Listing 3.3

In the response, the instances of `Probe` have been replaced by instances of `DecimalResult`[2]. The instances of `DecimalResult` are populated with the computed results by the DQL Connector and can be returned directly to the user for further processing and analysis. As this illustration observes only the communication between the DQL QEE and the DQL Connector, further tasks like the calculation of aggregates are not part of the examples. In Section 5.3.4 the process for aggregate calculation and their representation in the Mapping Meta-Model will be described.

5.3. Execution of Queries and Component Interactions

5.3.1. Registration and Lookup Process of the Connector Registry

OSGi allows to assemble components consisting of interfaces and corresponding implementations. These components are encapsulated within OSGi Bundles and can be deployed to an OSGi run-time environment [OSG12, p. 217]. Following the system architecture description for DQL Connectors in Section 5.1.1.4, for each supported performance meta-model family a distinct DQL Connector implementation exists. These Connector implementations are encapsulated as separated OSGi Bundles. Although multiple component definitions within a single OSGi Bundle are possible, the description following assumes each OSGi Bundle to offer a DQL Connector is only exposing a single component.

To provide the functionality contained in OSGi Bundles on demand, OSGi allows to expose components as *Declarative Service* to the run-time environment and manages dependencies [OSG12, p. 215 ff.]. Once a service is declared, it can be looked up through the OSGi Service Component Registry (SCR). The OSGi SCR is responsible for the lifecycle management of components. The lookup of components provides a result of the type `ServiceReference` that can be used to create an instance of the implementing class of the requested interface. Both are declared explicitly by the component. For each DQL Connector, the `ConnectorProvider` interface is exposed to the OSGi SCR.

The OSGi SCR is not capable to specify and index services by additional properties, e.g. a key represented as `String`, to identify different services for the same implemented

[2]The attribute `valid` is omitted for space reasons.

interface, e.g. the `ConnectorProvider` interface. Thus an OSGi run-time having multiple DQL Connectors deployed would not offer an option to select an appropriate Connector for the requested performance meta-model family. Instead each service providing the `ConnectorProvider` interface would have to be tested for support of the performance meta-model family requested and if the corresponding Connector is found an instance can be created. In case of multiple Connectors and frequent queries, this effort would not be reasonable and cause a significant overhead.

To work around this limitation, the DQL Connector Registry exists as introduced in Section 5.1.1.5. Through the Connector Registry the interface `ConnectorRegistry` is exposed together with a corresponding implementation. To realize a registration process for Connectors, the Connector Registry declares a 1-to-n reference to all services providing the `ConnectorProvider` interface. Through the OSGi Declarative Services, controlled through an event strategy, the registration of all DQL Connectors is initiated [OSG12, p. 220 ff.]. The OSGi SCR then triggers method calls on the `ConnectorRegistry` interface by events for binding, updating and unbinding a DQL Connector. Currently only the binding and unbinding is supported by the DQL Connector Registry.

An partial illustration of the process to register and obtain instances of a DQL Connector by using the Connector Registry is shown in Figure 5.10. The process can be split into three phases. In *Phase 1* the OSGi Declarative Services are recognized during the deployment of Bundles and registered at the OSGi SCR. The registration of the DQL Connector triggers the creation of the DQL Connector Registry and binds the Connector. In *Phase 2* the DQL QEE processes a query and tries to perform a lookup of a Connector. To find the Connector, the `ServiceReference` of the DQL Connector Registry is requested, which is used to obtain an instance of the service. Finally in *Phase 3* the `ServiceReference` and instance of a DQL Connector (i.e. supporting the requested performance meta-model family and query class) is acquired and the actual query processing can start.

To simplify lookups of DQL Connectors by their performance meta-model family and supported query class at the Connector Registry, an index data structure is used. When a new Connector is bound to the Connector Registry, the Connector is put into a two stage index, i.e. a key-key-value data store. The implementation of the index is realized using a `java.util.Map`. First using the performance meta-model family identifier as key referring to another instance of `Map`. The second `Map` contains an entry for each query class for which an implementation for this performance meta-model family exists. The value in the second `Map` instance is a `ServiceReference` that can be used to finally obtain an instance of the `ConnectorProvider`.

The resulting type of the index in the Java implementation is `Map<String, Map<Class, ServiceReference<ConnectorProvider»`. The lookup of a particular `ConnectorProvider` can be realized through a two-fold lookup in subsequent `Map` instances. In case of undeploying, i.e. unbinding, a DQL Connector, the Connector Registry can remove the entries from the data structure. In order to control the registration process, several additional properties exist in the meta-information of the component descriptor of a Connector. The process ensures a reasonable overhead even in case of frequent query request rates and a high number of available DQL Connectors while it ensures to be conforming to the dynamics of the OSGi run-time.

5.3.2. Execution of Model Structure Queries

Executing Model Structure Queries involves several layers of the DQL system architecture. The interpretation and control flow to execute Model Structure Queries will be outlined in this section.

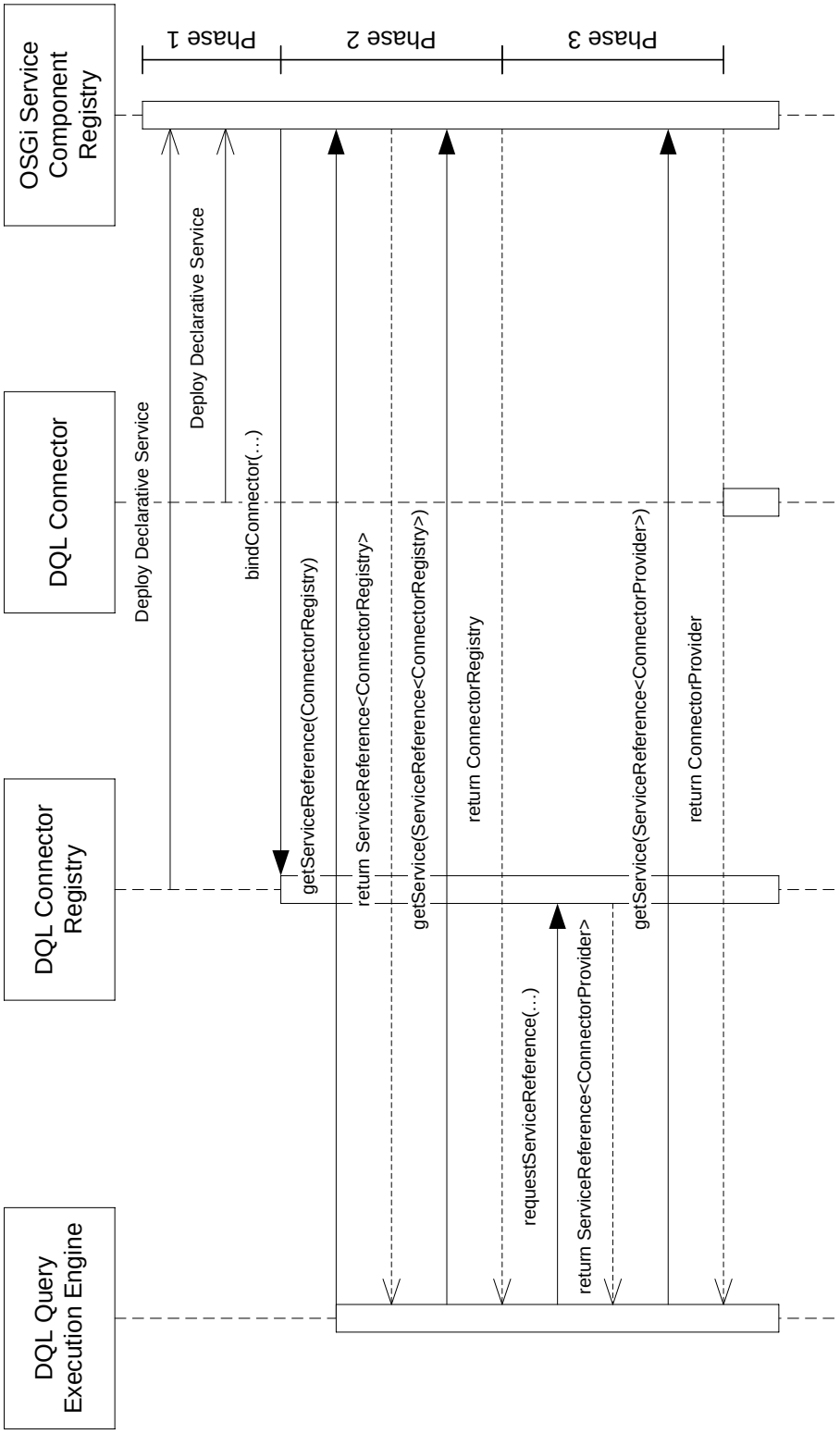


Figure 5.10.: Interaction of the DQL Query Execution Engine with the DQL Connector Registry

The involved components and implementation parts to execute a Model Structure Query being outlined in this section are:

- **DQL QEE → Implementation of the External Interface:** Implementation of an interface exposed to users to execute queries.
- **DQL QEE → Model Structure Query Interpreter:** Responsible for pre-processing and validating queries, performs transformation of queries into instances of the Mapping Meta-Model.
- **DQL Connector → Model Structure Query Connector:** Executes the query based on the information found in the instance of the Mapping Meta-Model, is specific to a performance meta-model family and utilizes external tools if necessary.

Figure 5.11 shows the conceptual control flow for processing a Model Structure Query and the involved parts of DQL. After a request is received by the implementation of the external interface of the QEE, it is delegated to an interpreter for its query class, i.e. the Model Structure Query Interpreter. As the availability of Connectors depends on the current state of the OSGi run-time and the declaration of services (see Section 5.3.1), the interpreter tries to find an adequate Connector to process the query. If no compatible Connector is found, the processing is aborted. Otherwise the processing can be continued and more compute-intensive tasks for preparing the query for processing can be performed.

The compute-intensive tasks performed are interpreting the query first, which is a task directly depending on the actual type of Model Structure Query issued, e.g. `LIST ENTITIES` in contrast to `LIST METRICS`, and, therefore special for each type of Model Structure Query. To delegate the execution of the request to the Connector instance created, it is encapsulated as instance of the Mapping Meta-Model. An illustration of the different instances of the Mapping Meta-Model for executing Model Structure Queries can be found in Section 5.2.3. After requesting the Connector to execute the query, the performance meta-model specific processing of the query is started. Specific processing involves accessing the performance model instance and evaluating it.

The processing of a request consists of multiple steps within the Connector. First the request needs to be processed, which involves accessing the model instance and to check for availability of all required information. The actual *solving process* depends on the used performance meta-model family and might involve an additional component for model solving. An example for an additional component could be a technology for querying model instances for model entities, e.g. EMF ModelQuery[3] in case of an EMF-based performance meta-model family like PCM or DMM. After completing the execution of the external solver, the results can be stored in a response using the Mapping Meta-Model and are returned to the user.

The process shown in this section is the generic approach for executing Model Structure Queries. The different Model Structure Queries defined as children of ListQuery require no additional processing at the moment. Later extensions of the Model Structure Query class could provide filtering mechanisms, that could be realized by additional processing steps implemented while returning results to the user. The abstraction layer introduced through the Mapping Meta-Model allows to operate by using generic processing steps among the different queries realized in the Model Structure Query class.

5.3.3. Execution of Performance Metrics Queries

To advance from the previous section, this section will outline the interaction between layers of the DQL QEE and a DQL Connector to execute Performance Metrics Queries. Contrary to Model Structure Queries, Performance Metrics Queries usually involve complex

[3] `http://www.eclipse.org/modeling/emf/?project=query`

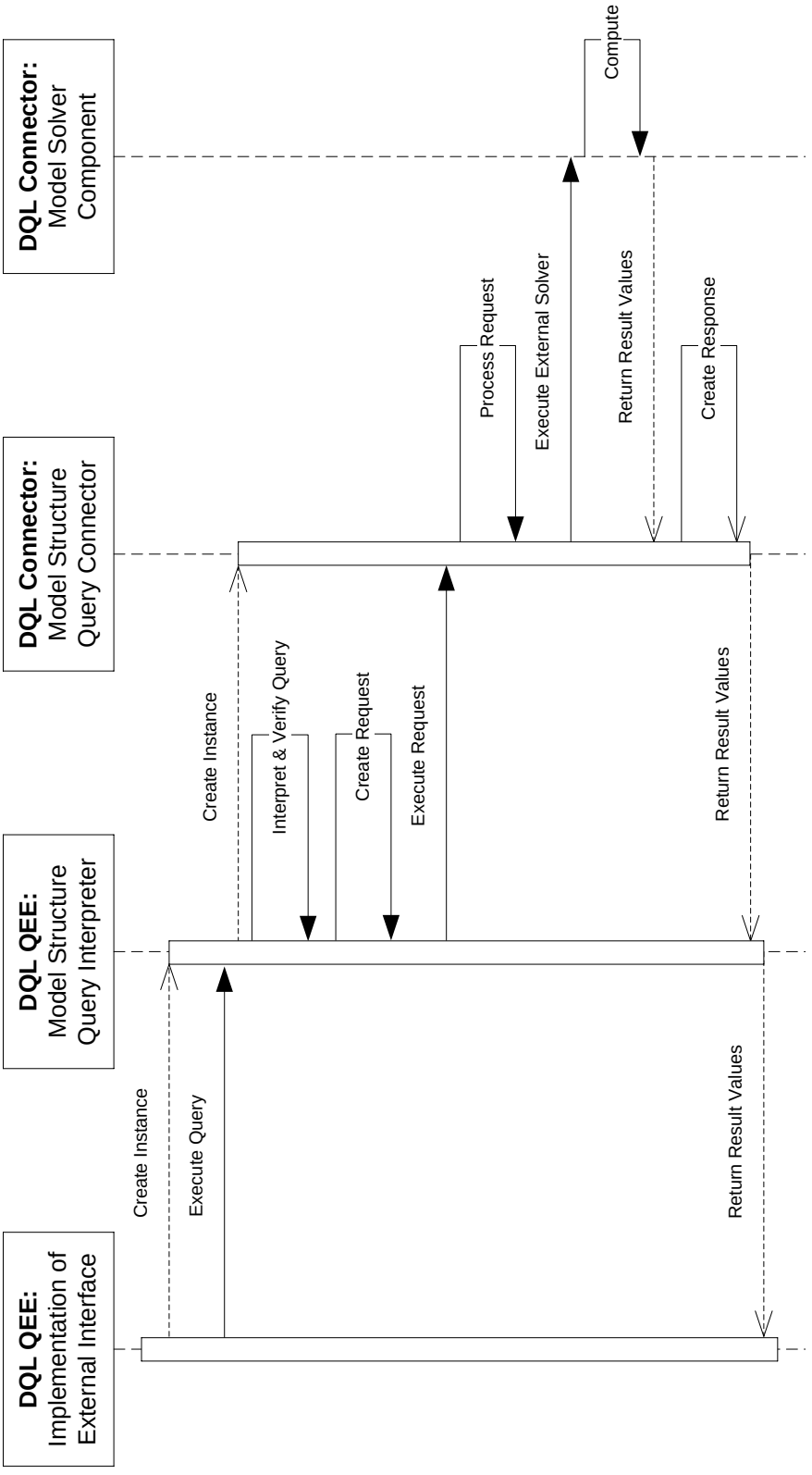


Figure 5.11.: Interaction between Software Components during the Execution of a Model Structure Query

model transformations and compute-intensive simulations that take place at the Connector layer.

The involved components and implementation parts to execute a Performance Metrics Query being outlined in this section are:

- **DQL QEE → Implementation of the External Interface:** Implementation of an interface exposed to users to execute queries.
- **DQL QEE → Performance Metrics Query Interpreter:** Responsible for pre-processing and validating of queries, performs transformation queries into instances of the Mapping Meta-Model, parameterizing the Connector.
- **DQL Connector → Performance Metrics Query Connector:** Executes the query based on the information found in the instance of the Mapping Meta-Model, is specific to a performance meta-model family and utilizes external tools if necessary.

Figure 5.12 illustrates the processing of a Performance Metrics Query for a Constrained Query, i.e. a SelectQuery containing a ConstraintClause, with an additionally defined temporal dimension through an ObserveClause. Extending this query towards an DoF Query using a DoFClause would cause additional operations to be executed being omitted in Figure 5.12 as they would not necessarily use additional layers, but degrade the overview of interactions in the figure.

The execution of the query starts by initiating the lookup process for a suitable DQL Connector for query execution taking place before the query is interpreted. As part of the subsequent interpretation process, the ConstraintClause and the ObserveClause are evaluated and the settings are propagated to the Performance Metrics Query Connector. To evaluate the ObserveClause additional time calculation steps are necessary as described in Section 5.3.5.

For both clauses, ConstraintClause and ObserveClause, the information obtained contains no structural properties of the performance model instance. As the Mapping Meta-Model is intended as a abstraction layer for the structure of a performance model instance, the information is not embedded as part of the Mapping Meta-Model within the request. To delegate the information to the Connector, `set`-methods of the Performance Metrics Query Connector are used.

The subsequent steps are related to the creation of the request through the Mapping Meta-Model. Before the request is executed by the Connector, additional steps are required for computing aggregates that may be part of the request. These steps are described in Section 5.3.4. In case an aggregate is requested, certain conditions imply modifications of the original request, but during a post-processing step all modifications that are applied implicitly are reverted before the results are passed back to the user.

The process of calculating performance metrics is specific for each DQL Connector implementation. As performance meta-model families differ significantly no general assumption on the calculation process can be made and is subject to the implementer. As for the computation process no assumptions are made at any layer of DQL, neither any assumptions are made for the applicability for DQL Connector settings as constraints or temporal dimensions. For these settings even at the Connector level no general assumption on the impact of settings can be made, as they depend on the underlying performance model instance.

5.3.4. Calculation of Aggregates on Performance Metrics

In Section 3.5.5 language elements for calculating aggregates on top of performance metrics are introduced. Aggregates in DQL are part of the language and are not being calculated

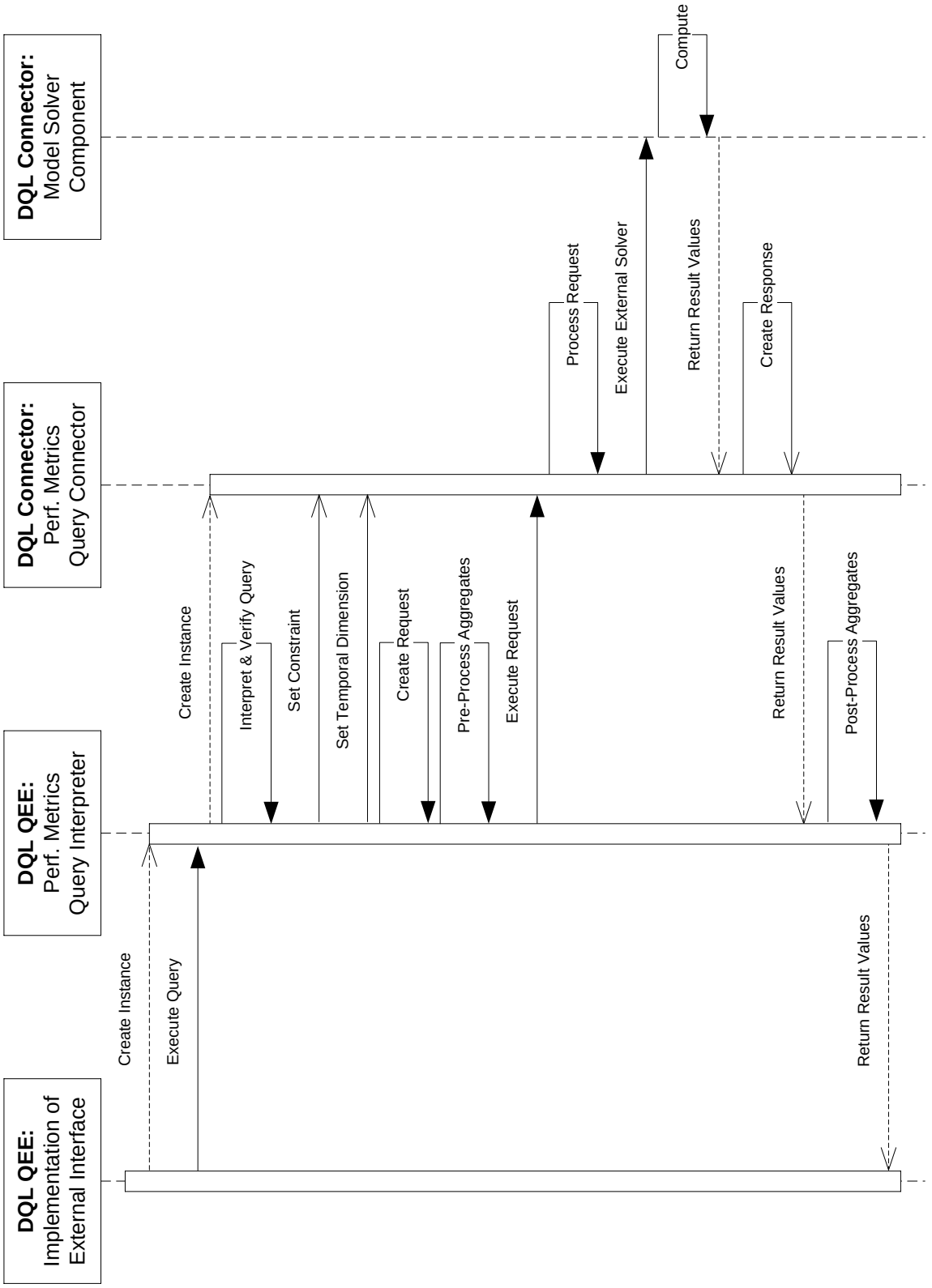


Figure 5.12.: Interaction between Software Components during the Execution of a Performance Metrics Query

by requesting aggregations of performance metrics through functionality of the DQL Connector. Instead, aggregates are calculated at the DQL QEE layer and their influence on performance metrics calculation at the DQL Connector is as minimal as possible. The most significant influence are requests of the QEE to calculate additional performance metrics implicitly, among other performance metrics requested through the query, in order to compute desired aggregates. For users this process is transparent. This section is an addition of Section 5.3.3, where the execution of queries from the class of Performance Metrics Queries is outlined.

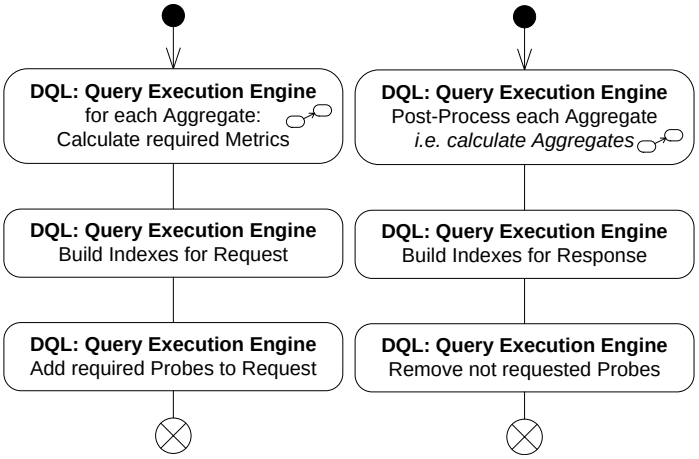
The process of calculating aggregates is integrated directly with the process of executing common Performance Metrics Queries. The process is initiated during the evaluation of the MetricReferenceClauses within the `PerformanceMetricsQueryInterpreter`. All found aggregates are registered in a central data structure for an additional *pre-processing* step before submitting the request to the Connector for execution. A finalizing *post-processing* step is executed after the Connector has completed its work in order to calculate the values of all aggregates. The feature is implemented orthogonally to the remaining processing of Performance Metrics Queries, i.e. the Connector requires no special functionality to offer support for aggregates.

Figure 5.13a shows the steps for calculating aggregates as part of the pre-processing phase using the `AggregateController` implementation of the QEE. At first, for each aggregate, the required performance metrics to compute the aggregate are stored in a index data structure. The type of the index is `Map<String, Set<String>`. The key in the `Map` is the identifier of a model entity, the `Set` contains all required metric names of the entity in order to compute the requested aggregates. Using this index, all metrics required additionally being found during the pre-processing step can be tracked and merged among the other aggregates requested. Thus the pre-processing ensures that no duplicated requests for additionally inserted performance metrics can occur.

As aggregates are implemented orthogonally and the Connector is not aware of requested aggregates, the pre-processing step in the QEE has to request additional performance metrics for computing the aggregates implicitly. In order to perform an implicit request, the performance metrics required additionally are merged with the instance of the `EntityMapping` that is already available and is created through the usual processing of requests in the QEE. The pre-processing consists of two additional pre-processing steps that are necessary to extend the set of performance metrics within the originating request (M_r) with additional performance metrics ($M_{a'}$). The additional performance metrics are not yet part of the request and stem from the set of performance metrics required for aggregate computation (M_a). The resulting set $M_{a'} = M_a \backslash (M_a \cap M_r)$ of performance metrics is added as common requests for performance metrics among the explicitly requested performance metrics to the original request and can be delegated to the Connector for computation. Thus the Connector implementation is not required to be changed to support calculating metrics for aggregate computation.

To add additional performance metrics efficiently, the request is indexed as the additional performance metrics are. To detect missing metrics, the additionally required metrics are iterated and if metrics are missing in the request, they are added. The operations on the data structures are realized through the operations available for `Map` and `Set` with the corresponding implementations based on hash functions allowing to access contents efficiently.

After the computation of performance metrics through the Connector is finished and the `EntityMapping` used as response is available, the post-processing of aggregates as shown in Figure 5.13b is started. Using the response of the Connector, all aggregates, that have been recognized during pre-processing, are to be calculated and inserted into the `aggregates`



(a) Pre-Processing Steps (b) Post-Processing Steps

Figure 5.13.: Steps for Pre- and Post-Processing of Aggregates in addition to Figure 5.12

containment of `EntityMapping`. The calculation of aggregates is based on the software library Apache Commons Math[4], which offers support for the computation of statistical metrics, and in case of trivial computations on self-implemented classes for computing the aggregate. The available aggregates supported by DQL are exposed through the StatisticalAggregateType grammar rule in queries. Future extensions can reuse the existing infrastructure to offer additional aggregates by extending the StatisticalAggregateType at the language level and providing a suitable implementation of `AggregateCalculator` in the QEE.

Complementary to adding performance metrics required implicitly to the requesting `EntityMapping`, the post-processing removes not explicitly requested performance metrics from the `EntityMapping` instance for responding. If the performance metrics would be kept in the response, users might be served with a result that was not requested and consider it as malicious. For the removal of performance metrics not requested explicitly, the index structure of the pre-processing containing all missing performance metrics, $M_{a'}$, is reused. In order to perform the removal efficiently, the response is indexed with two inverse indexes.

The index of all calculated performance metrics in the response is of the type `Map<String, Map<String, Probe»`, where the first key is the `identifier` of an `Entity` in the Mapping Meta-Model and the second key is the `metricName` of a `Probe`. This `Probe` is referenced by an `Entity`, with a final reference to the instance of `Probe` as value. At run-time the response does not contain instances of the type `Probe` but `DecimalResult` to transport results. As `Probe` is the super-type and no explicit downcasting is necessary, the implementation uses the parameterization through the type `Probe` which is preferable as additional result types that might be added in future are already supported. Using this index together with the index of additional performance metrics, the corresponding `Probes` of performance metrics added implicitly can be found efficiently. In order to detach these `Probes` from their `Entity` instances, another index of the type `Map<String, Entity>` is utilized, which uses the `identifier` of an `Entity` as key and the `Entity` as value. The removal process can use this index to call the corresponding methods for removing a `Probe` from an `Entity` at the corresponding `Entity` instance.

To present the calculated aggregates to the user, the results are stored in the `EntityMapping` used as response. For storing the response, the `probes` attribute of the `Aggregate` type is used and the `Aggregates` are stored in the containment `aggregates` of `EntityMapping`.

[4] `http://commons.apache.org/proper/commons-math/userguide/stat.html`

For meta-information, e.g. in case of the `PERCENTILE` aggregate (see StatisticalAggregateType), that are required to calculate the aggregate, the `properties` attribute of `Aggregate` is used. Although the Mapping Meta-Model shall only represent a view on the structure of the performance model instance used for analysis, this extension is reasonable. Aggregates do not represent the structure of the performance model instance, but are directly based on entities representing the performance model instance structure. Furthermore, from a technical view, the embedding of higher-order information within the Mapping Meta-Model allows to present a coherent view on a performance meta-model instance under the angle of the query. Excluding the results of aggregate calculation would have made another approach for the representation of metrics necessary.

In summary the process of calculating aggregates could be implemented as orthogonal feature in the class of Performance Metrics Queries. The approach is independent of capabilities of Connectors as long as Performance Metrics Queries are supported and might can be leveraged to be used for calculating aggregates based on the evaluation of DoFs. The implementation of pre- and post-processing of aggregates is implemented to support even huge amounts of aggregates and performance metrics used in aggregates by using adequate data structures and indexing methods. More comprehensive queries might be a result of the usage of DQL in usage scenarios generating queries by machine users, e.g. by a Performance Testing System (see Section 3.1.2.2) or a Monitoring and Resource Management System (see Section 3.1.3.2).

5.3.5. Absolute Calculation of Time Specifications

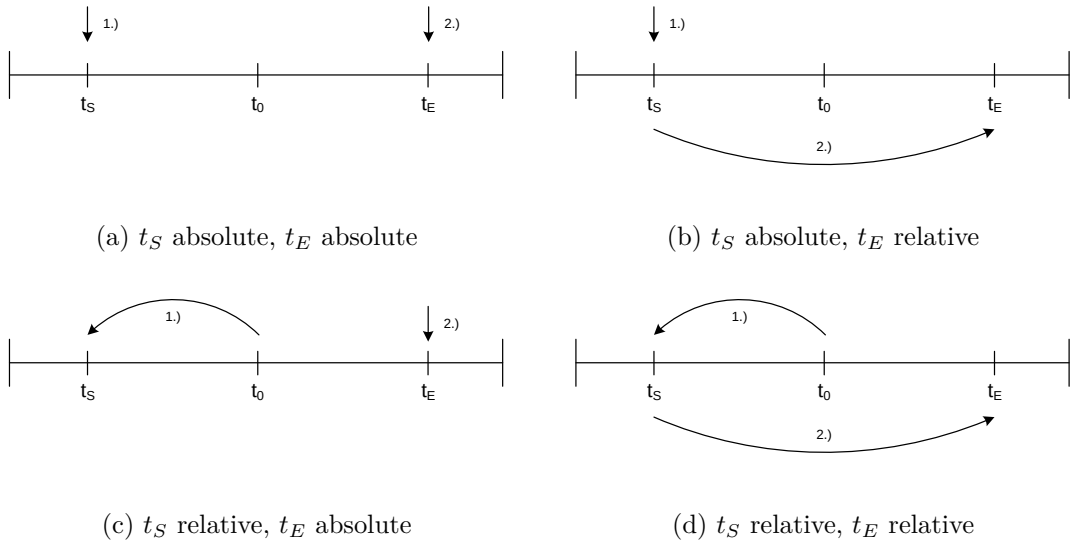
The usage of time in the ObservationClause with the referenced grammar clauses ObserveRelativeClause and ObserveBetweenClause and further child clauses raise the need for a definition of the calculation of time points in the DQL. As the definition of interpreting time is not part of the language design, the decision for time calculations takes part in the implementation of DQL. For calculating absolute points in time, the interpretation of the temporal dimension needs to be realized within the DQL QEE in order to provide consistent results. Additional, Connector-dependent interpretations, can be provided by a DQL Connector and are subject to be specified by the Connector implementer.

The relevant language parts addressed in this section are used to compute time frames. A time frame has an absolute point in time for its start and end. As relative time expressions are allowed, the absolute points in time to define a time frame need to be calculated. For defining rules to calculate absolute points in time, the following expressions will be used:

- t_0 is the present point in time.
- t_S/t_E is the absolute point in time of the start/end of a time frame.
- d_S/d_E is the relative time duration specified as the starting/ending point in time.
- T is the time frame spanned by t_S and t_E.

The temporal ordering of points in time assumes $t_S < t_E$, i.e. the absolute point in time t_S is earlier than t_E. No assumption is made on a constraint on t_0 and its relative ordering to t_S or t_E. t_0 is only required for obtaining a reference date to calculate relative date specifications in case no other reference date can be used. Thus the position of t_0 in Figure 5.14 does not reflect the temporal ordering of t_0, t_S and t_E.

Figure 5.14 illustrates the different cases for calculating time frames. The cases shown correspond to the time calculations performed in the classes `ObserveBetweenInterpreter` and `ObserveRelativeInterpreter` of the DQL QEE. The calculation of the time frame T and the interpretation of t_S and t_E is implemented with the following semantics:



(a) t_S absolute, t_E absolute

(b) t_S absolute, t_E relative

(c) t_S relative, t_E absolute

(d) t_S relative, t_E relative

Figure 5.14.: Reference Points for Time Specifications

- Figure 5.14a: T is defined directly with t_S as starting point in time and t_E as ending point in time.
- Figure 5.14b: T is defined by using t_S as starting point in time and t_E is specified as $t_E = t_S + d_E$, i.e. the relative duration specified as end is added to the absolute start of T.
- Figure 5.14c: To define T at first t_S has to be calculated as $t_S = t_0 + d_S$ and the end of T can be directly specified as t_E. Otherwise as the visualization suggests, t_S does not need to be a point in time before t_0. The calculation of t_S allows both, i.e. negative and positive specifications of duration.
- Figure 5.14d: As for the previous case at first t_S has to be calculated as $t_S = t_0 + d_S$, which is then reused to calculate t_E as $t_E = t_S + d_E$. After t_S and t_E have been calculated, T is defined.

The steps for calculating and the case-dependent definition of reference points for calculating absolute points in time from relative specifications allow to map all possible combinations of time specifications that can be expressed through the DQL.

5.3.6. Configuration of Degrees-of-Freedom Evaluation

The configuration of exploring DoFs is two-fold and takes place at the query language level. Through the DoFClause a user can (i) configure the process-specific settings for the DoFs exploration and (ii) specify the parameter space for each DoF available in the performance model instance used. Further configuration settings might be based on DQL Connector-specific configurations, e.g. configuration files within OSGi Bundles, that are not part of DQL.

The process-specific settings for DoFs evaluation are specified through the non-terminals ExplorationStrategyClause, ConstraintClause and WithClause. For each of these clauses, a corresponding pair of `get`/`set`-methods exists in the `PerformanceMetricsQueryConnector` interface. The actual values of settings and their interpretation is subject to the Connector. Especially for the interpretation of the WithClause and to construct an advanced usage scenario of DoFs evaluation with an additional exploration controller see Section 6.4. The usage scenario introduced in the referenced section gives an outlook for the implementation

of a software component for systematically exploring DoFs integrated with a Connector to form a control loop.

For specifying the individual parameter space of a DoF within a VaryingClause, the ConfigurationPropertiesClause can be used. As linear parameter space expansion and setting a set of fixed values for a DoF might be common use cases, the DoFVariationClause exists. The DoFVariationClause and its children IntervalVariationClause and ValueVariationClause are abbreviations of the ConfigurationPropertiesClause, to which it is transformed in the DQL QEE. An example is shown in Listing 3.6.

The ValueVariationClause is used to define a set of values of the query level that shall be applied to a DoF. Thus the ValueVariationClause is a simple example for eliminating the need to modify a performance instance manually to change individual entity values for retrieving performance metrics using an external tool supplementary to DQL. If a ValueVariationClause is specified, it is transformed into the following set of property keys and values before it is submitted to the Connector:

- `function` with the fixed value *fixed*.
- `value` with one or multiple integer values separated by a ,.

A Connector should use the values provided in the `value` property to set each value as a value for the DoF, which will be used to perform the calculation of performance metrics. Thus the amount of supplied values for varying a DoF has direct influence on the amount of computations required and results returned.

The IntervalVariationClause represents an iteration of integer values spanning an linear interval of discrete values. It is corresponding to a `for` loop found in programming languages. The IntervalVariationClause is transformed into the the following property keys of the ConfigurationPropertiesClause:

- `function` with the fixed value *iterate* to indicate how to interpret the other properties.
- `start` (s) with an integer value as start of the interval to be spanned.
- `end` (e) with an integer value as end of the interval to be spanned.
- `step` with an integer value to be added to the current value of the iteration.

Using these properties, a Connector implementation should interpret the properties corresponding to the following `for` loop: `for(int i = start; i < end; i += step) { /* ... */ }`. The interval specified through the IntervalVariationClause is formally defined as $(s, e] \in \mathbb{Z}$.

As the DoFs specified in a VaryingClause represent information that are directly related to the structure of a performance model instance, they are eligible to be represented through the Mapping Meta-Model directly. An instance of `EntityMapping` allows to store and represent DoFs as the type `DoF` as shown in Figure 5.6. Furthermore as the `DoF` type is derived from `ExtendedEntity` it allows to store an instance `Property` as attribute, which is used to represent the meta-information contained in the ConfigurationPropertiesClause. Following this approach the QEE can use the Mapping Meta-Model to forward all necessary and structurally relevant information to the Connector.

Vice-versa the Mapping Meta-Model can also be leveraged to represent the setting of a DoF when the results of a query are returned. The setting of a DoF can be represented as `DecimalResult` attached to a `DoF`. Thus a coherent view on the performance metrics returned through an instance of the Mapping Meta-Model and the current setting of DoFs

can be ensured. Through the coherent view, a user can recognize the setting of DoFs easily without guessing the order of evaluating DoFs. QEE and the corresponding Connector can leverage this effect to perform optimization during execution, i.e. to parallelize executions of queries in the parameter space, without taking care of a special ordering of executions.

5.4. Eclipse-based User Interface for Performance Analysis

5.4.1. Query Editor and Execution Environment

Based on the source code generated by Xtext, an editor with syntax highlighting and checking support is available. The editor allows users to write queries in DQL within the Eclipse IDE and is a foundation for the implementation of a UI for DQL. The main focus for UI enhancements is on the execution of queries, the integration with the Eclipse platform and the implementation of content assist hints for the query editor.

For a tight integration with the Eclipse platform, the ability to execute DQL queries within the IDE is a mandatory foundation. To integrate DQL and to launch queries, two steps are necessary: (i) Implement `org.eclipse.debug.ui.ILaunchShortcut` in order to provide the launch support as it is commonly available for e.g. Java programs through using the *Run Dialog* and (ii) implement the abstract class `org.eclipse.core.runtime.jobs.Job` in order to delegate the execution of a query to the Eclipse platform.

At the current implementation level of the DQL in the Eclipse IDE no additional settings are required in order to execute a query. The Run Dialog is therefore populated with a default dialog window derived of the class `org.eclipse.debug.ui.AbstractLaunchConfigurationTabGroup` presenting a familiar view to the user. The execution of jobs is set to be scheduled in the background for long-running jobs. The value is specified statically using the constant `Job.LONG`.

Xtext provides a mechanism to implement content assist for the implementation of a language. The code generation of Xtext provides an abstract class that can be implemented to provide content assist for individual grammar rule combinations for allowed language constructs. The current implementation of content assist for DQL allows to complete statements with the identifiers and available metric names of model entities. To realize the content assist of identifiers, in the background DQL queries are generated and executed to examine the model instance. The results are then displayed and a user may choose between alternatives. The content assist is also usable to complete identifiers in case a user provided only a partial identifier.

The resulting DQL Editor is embedded in the Eclipse IDE as shown in Figure 5.15. The upper half of the screenshot shows the DQL Editor generated through Xtext. The query shown has special formatting for a query and allows to use the content assist to complete parts of the query by suggestions. Through the launcher mechanism of Eclipse, the query can be submitted to the DQL QEE for execution.

5.4.2. Presentation of Query Results

To display query results, a view for the Mapping Meta-Model is available. The view is based on the class `org.eclipse.jface.viewers.TableViewer` and visualizes instances of the Mapping Meta-Model as flat table. The view is embedded in the Eclipse IDE and users may choose where to display query results.

The implementation is shown in the lower half of Figure 5.15 displaying the results of the query execution. As the query contains DoFs, multiple tabs are visible displaying the different instances for each combination of DoF parameter setting. The table displays the

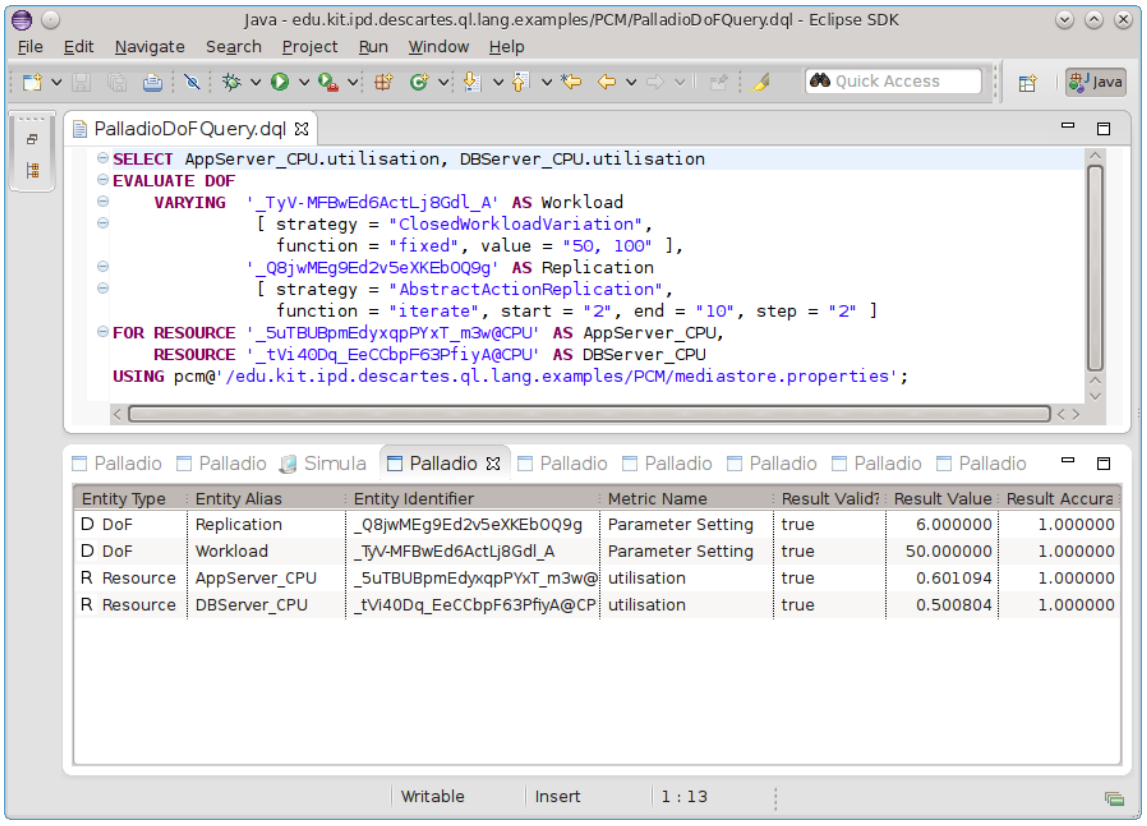


Figure 5.15.: Screenshot of the Eclipse IDE showing a DQL Query for PCM

for each available instance of `DecimalResult` one row showing all relevant information, e.g. the `Entity` it belongs to.

As the Mapping Meta-Model is currently the only data structure used for representing the results of queries, i.e. serving for all query classes, results from all query classes can be represented through instances of this view. For later revisions, the view can be extended to support more features, e.g. on demand filtering on a column level.

6. Evaluation of Design and Implementation

This chapter presents an evaluation of the DQL approach. It evaluates the design decisions from Chapter 3 and the implementation approach from Chapter 5 by investigating approaches for the interaction of DQL with established scientific approaches. In Section 6.1 an evaluation of a DQL Connector implemented for PCM is presented showing DQL controlling PCM. Section 6.2 shows a conceptual approach to integrate DQL with KLAPER for analyzing performance model instances through using intermediate models. A view on the capabilities of temporal dimension of DQL is presented in Section 6.3 showing an approach to extend the Mapping Meta-Model into a PDR. The chapter is finished with an outlook in Section 6.4 to a prospect usage scenario of Software Performance Cockpit (SoPeCo) in a control loop with the DQL approach to conduct efficient experimentation with DoFs.

6.1. Controlling the Palladio Component Model (PCM)

6.1.1. Introduction to the Palladio Component Model

PCM is a modeling language for the architecture of CBSE-based software systems and their deployment on resource landscapes to predict the performance of systems. The approach addresses the needs of users during design, development and maintenance phases of software systems [RBB+11]. As CBSE encourages the reuse of generic components in arbitrary environments, each component represents a set of provided and required interfaces together with business logic. Components are intended to be used as black box building blocks as only the interfaces are exposed. To ensure an acceptable level of QoS or to satisfy SLAs, it is important to predict the performance of these independent components after their assembly. Using these predictions, the PCM approach supports the CBSE process in a way to allow architectural decisions for the selection of components and their composition as early as possible. A detailed and illustrated description of PCM approach can be found in [BKR09].

The Palladio Bench[1] is an Eclipse-based set of tools for modeling instances of PCM and to execute simulations of these models. The tool allows to define model instances through using graphical editors with an UML-based graphical syntax. When a simulation run is

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

completed, the results obtained during simulation can be accessed through various representations. The modeling part of the Palladio Bench is realized by means of EMF and related tools.

For the evaluation of the DQL implementation, PCM is a valuable target. On the one side, PCM has already been used in case studies of real-world systems, e.g. enterprise storage systems in [HBR+10] and large-scale e-mail systems in [RBKR12], and on the other side PCM is available as an approach for SPE from academia with a working software stack and validated examples. The examples allow to compare the usage of the Palladio Bench directly and DQL as interface to the Palladio Bench. The Palladio Bench and PCM are therefore viable evaluation targets for DQL. Especially the internal data structures, the Mapping Meta-Model in DQL, and the interfaces for the communication between the DQL QEE and DQL Connectors can be evaluated in-depth.

6.1.2. Mapping the Palladio Component Model

The integration of PCM with DQL requires to provide suitable data structures to store all relevant information. From the viewpoint of DQL, especially the Mapping Meta-Model must be able to transport the necessary information to reference model entities from PCM, to specify which metrics to calculate and to return all computed information from PCM to the user.

PCM consists of various sub-packages, that are composed to form an architecture-level model of a software system and a resource landscape. The behavior of components is specified in Service Effect Specifications (SEFFs). For performance predictions, especially the Resource Demanding Service Effect Specification (RDSEFF) is important. A RDSEFF contains, besides of control flow-relevant information, the expected resource consumption of actions executed in components [BKR09, RBB+11].

As described by [Mer11, p. 51] the resulting PCM model entities for which response times are provided through the simulation of PCM instances, are `EntryLevelSystemCalls`, `ExternalCallActions` and `UsageScenarios`. These entities are mapped to `Services` in the Mapping Meta-Model. By convention a `Probe` with the metric name *responseTime* will be provided on request for these entities.

Within the referenced RDSEFF of these model elements, the resource demand is mapped to resources. Detailed descriptions of the modeling of a RDSEFF can be found in [RBB+11, p. 48 ff]. Stated by [Mer11, p. 51] the metrics available for active resources are *demandedTime* and *utilization*. The current mapping approach maps all `ResourceContainers` and `LinkingResources` as `Resource` in the Mapping Meta-Model.

6.1.3. Run-Time Scenario and the MediaStore Example

Currently the Palladio Bench does not offer an unified API for controlling and analyzing simulations. To control the simulation process, the Palladio Experiment Automation[2] is maintained as an incubating project for the Palladio Bench. The implementation of this interface is based on the work in [Mer11] to conduct automated experiment runs.

The interface expects a model instance of the PCM as input and allows to set various simulation-dependent parameters. To automate experimentation, the API allows to specify model elements that shall be varied in consecutive simulations. The variations are realized through model transformations of the originating PCM instance. To ensure not to violate the model semantics, the variations are implemented context-dependent with specialized

[2] `http://sdqweb.ipd.kit.edu/wiki/Palladio_Experiment_Automation`

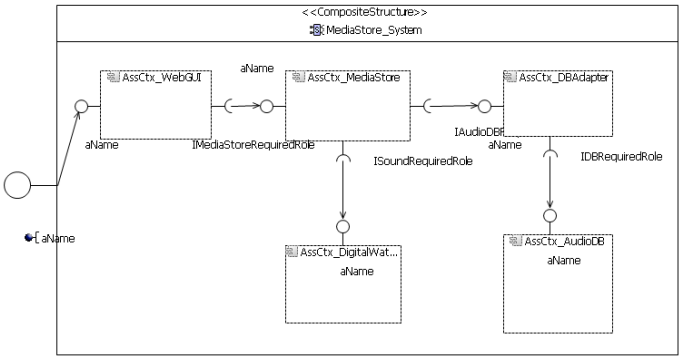


Figure 6.1.: Component Diagram of the MediaStore Example from [IPD12]

implementations for each model entity to be varied (e.g. for workload classes, loops or branches).

There exists no external API to retrieve results from simulations to direct performance metrics. The internal Sensor Framework, that is used to store computed results of sensors during the simulation, can be queried to retrieve the sensor measurements. Using the sensor measurements, performance metrics can be calculated manually. As the Palladio Bench is not designed to be used for automated analyses, this limitation is reasonable. Results presented through the DQL should be seen as exemplary as the results in PCM might differ and the implementation of the calculation of results from the Sensor Framework in DQL is only intended to be a workaround from the API limitations.

For evaluation and demonstration purposes, the Palladio Bench provides the *MediaStore*[3] example. The MediaStore is used to represent a web application consisting of multiple components. The application serves requests from customers to access media files stored in a repository. The components are allocated on a middleware layer with an additional database access layer. The middleware and the database component are allocated on different computing environments with exclusive hardware interconnected by an Ethernet link. Figure 6.1 shows a view on the components of the MediaStore. By using the mapping described in Section 6.1.2, the MediaStore is a suitable target for testing the implementation of a DQL Connector and to evaluate the DQL textual syntax and implementation.

6.1.4. Case Study with the MediaStore Example

To illustrate the features of the DQL Connector for the PCM, a case study for demonstrating the capabilities of DQL and the implementation of the DQL Connector for the Palladio Bench will be given. The case study uses the MediaStore example and demonstrates the usage of DQL Queries to control the Palladio Bench. The queries allow to automate simulations and to extract structural information from the different sub-models of the PCM.

6.1.4.1. Model Structure Queries

To start analyzing the MediaStore, a user would start to explore the model structure as in Listing 6.1. Using `LIST ENTITIES` all mapped PCM entities found in the MediaStore are presented to the user in the means of DQL either as `Resource` or `Service`. As being mandatory for later identification of entities, the `identifier` field in the Mapping Meta-Model entities contains the identifier from the PCM instance, i.e. information retrieved

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

from the `Identifier` class. Furthermore, where applicable, the `alias` field in the Mapping Meta-Model `Entity` is set to the value from the `NamedElement` class.

For performing the lookup process of model entities, the Connector accesses the model instances by issuing OCL queries. As a complete instance of the PCM consists of multiple sub-model instances, the `USING` keyword references a properties file serving as index to all additionally required information and model locations.

```
1 LIST ENTITIES
2 USING pcm@'mediastore.properties';
```

Listing 6.1: MediaStore Example for `LIST ENTITIES`

For the mapping of resources in the PCM, a workaround is required. The Sensor Framework provides only access to `Strings` built from different patterns to identify the results of the simulation. In case of a `ResourceContainer`, e.g. a server system or a VM from the resource landscape, that consists of multiple `ProcessingResourceSpecifications`, only the absolute identifier of the `ResourceContainer` is parsable. The different `ProcessingResourceSpecifications` are represented by their `NamedElement` alias and not the unique `Identifier`. For `LinkingResource` the situation is alike.

For reasons of simplicity the absolute identifier of the container, e.g. `ResourceContainer` or `LinkingResource`, is used and concatenated with the alias of the actual resource contained. Using this approach, the MediaStore example is usable without modification in this case study. Future changes of the MediaStore might lead to different, unpredictable results.

6.1.4.2. Performance Metrics Queries

After all relevant model entities in the PCM instance have been discovered, users can continue to explore structural information. Listing 6.2 shows how to retrieve the available metrics for PCM model instances. The shown identifiers contained in the `FOR` keyword represent the absolute identifiers from the PCM sub-model instances. As the Palladio Bench consists of graphical tools for manipulating the model instances, the identifiers are usually not visible. In case of a textual tool like DQL, using the aliases through the `AS` keyword allows an easier re-identification of identifiers for human users.

As the Palladio Bench offers no interface to test for metrics that are computable for model entities without analyzing simulation results, the available metrics are hard coded in the Connector based on the instance type of PCM model entity queried. The definition of available metrics is based on [Mer11, p. 51] and has been tested by reverse engineering the results obtained from the Sensor Framework.

```
1 LIST METRICS
2    (RESOURCE '_5uTBUBpmEdyxqpPYxT_m3w@CPU' AS AppServer_CPU,
3     RESOURCE '_tVi40Dq_EeCCbpF63PfiyA@CPU' AS DBServer_CPU,
4     RESOURCE '_tVi40Dq_EeCCbpF63PfiyA@HDD' AS DBServer_HDD,
5     SERVICE  '_u770YEhFEd2v5eXKEbOQ9g' AS svc1)
6 USING pcm@'mediastore.properties';
```

Listing 6.2: MediaStore Example for `LIST METRICS`

At this step users could gather enough structural knowledge of available model entities and computable metrics for these entities. Users can now start to issue performance queries

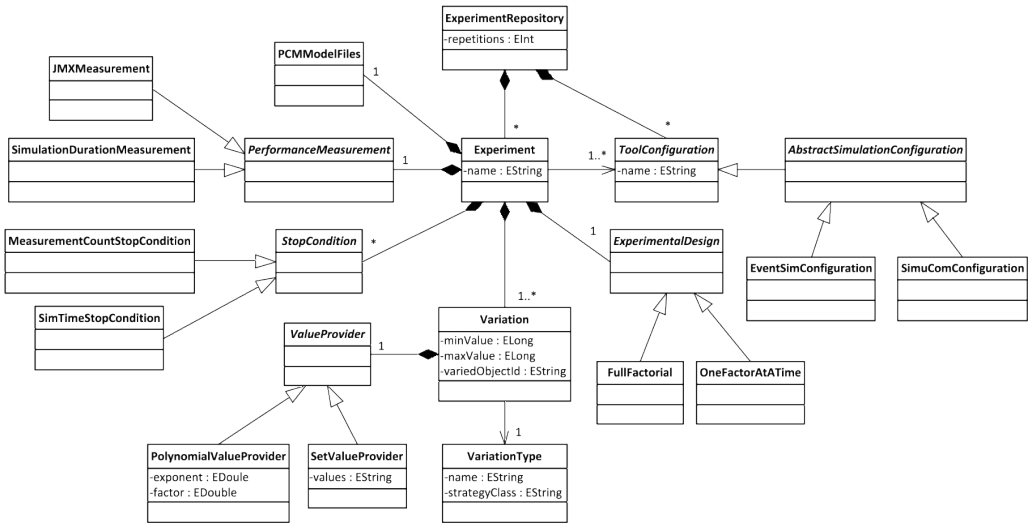


Figure 6.2.: Experiment Automation Configuration Meta-Model from [Mer11]

to obtain performance metrics computed through the simulation of the PCM instance. As mentioned in Section 6.1.3, the simulation process is controlled through the PCM Experiment Automation interface as described in [Mer11, p. 60 ff.]. The simulation results are cached on disk, therefore subsequent queries require no additional simulation. In case of changes to the model instance, the cached results need to be deleted manually.

An example for a basic performance metrics query is shown in Listing 6.3. The use of aliases in the `FOR` keyword allows users to use more convenient shortcuts in the `SELECT` keyword. To obtain the results, the Connector uses the Sensor Framework and accesses the stored simulation results on disk. As described previously, the access requires workarounds as the Palladio Bench is not designed for external access.

```
1 SELECT AppServer_CPU.utilisation, DBServer_CPU.utilisation,
2        DBServer_HDD.utilisation, svc1.responseTime
3 FOR RESOURCE '_5uTBUBpmEdyxqpPYxT_m3w@CPU' AS AppServer_CPU,
4     RESOURCE '_tVi40Dq_EeCCbpF63PfiyA@CPU' AS DBServer_CPU,
5     RESOURCE '_tVi40Dq_EeCCbpF63PfiyA@HDD' AS DBServer_HDD,
6     SERVICE  '_u770YEhFEd2v5eXKEbOQ9g' AS svc1
7 USING pcm@'mediastore.properties';
```

Listing 6.3: MediaStore Example for `SELECT`

As aggregations of metrics are supported by DQL, further calculations based on these metrics are possible, e.g. to calculate the mean utilization of a set of resources. The feature is part of the language and executed in the DQL QEE, therefore it is realized independently of the Connector.

6.1.4.3. Degrees-of-Freedom as Extension of the Palladio Bench

In [Mer11] further features of the Experiment Automation interface are explained. Besides of controlling the simulation engine of the Palladio Bench, also altering PCM model instances is possible. The Experiment Automation interface consists of a meta-model for the configuration of experiments as shown in Figure 6.2. Within this meta-model, the `Variation` type allows to specify how a specific model element shall be varied. Using a `VariationType` with semantic support of the model entity to be varied and a

`ValueProvider` to provide variation values, the meta-model allows to define a configuration parameter space.

Currently, the Palladio Bench offers no feature to define model parameter variations or, by means of DQL, it does not allow to specify DoFs. The Experiment Automation interface works around this limitation by modifying copies of the PCM sub-models and executing several independent simulation runs. Accessing the results is possible through the Sensor Framework, which allows to store multiple experiment runs within one experiment series.

In summary, using the model variations allows to implement DoFs in the DQL Connector for PCM. Hence, as described before, even if the results may not be accurate, the Connector would be usable as interface to the Palladio Bench for specifying the execution of an experiment series with varying parametric settings for model entities.

The resulting implementation of DoFs allows to perform at a first step to analyze which DoFs are available for model variation as shown in Listing 6.4. The PCM sub-models are queried through the use of OCL queries and available DoFs that comply with the Experiment Automation interface are listed.

```
1 LIST DOF
2 USING pcm@'mediastore.properties';
```

Listing 6.4: MediaStore Example for `LIST DOF`

In a second step, users can retrieve performance metrics while specifying which DoFs shall be varied. The Listing 6.5 shows the extension of the preceding Listing 6.3 with the `EVALUATE DOF` keyword. The query alters two DoFs independently resulting in a configuration space with a multiplicative dependency of factors and factor levels. Instead of a single result, DQL returns one instance of the Mapping Meta-Model for each combination in the configuration parameter space.

The variations are specified for model elements from PCM sub-models. The DoF *Workload* is the population of requests within the simulation, the *Replication* replicates executions of an `AbstractAction`. For each DoF a valid `VariationType` has been set as *strategy* in the DoF-specific properties. The `VariationType` implements all necessary logic to modify PCM sub-models in order to set the variation values for the model instances created. The implementation of `VariationType` is part of the Experiment Automation implementation. For both value specifications of the DoFs in DQL, the specifications are transformed into a compatible format to be used by the `SetValueProvider` from the Experiment Automation interface.

```
1  SELECT AppServer_CPU.utilisation, DBServer_CPU.utilisation,
2         DBServer_HDD.utilisation, LAN.utilisation, svc1.responseTime
3  EVALUATE DOF
4          VARYING  '_TyV-MFBwEd6ActLj8Gdl_A' AS Workload
5               [ strategy = "ClosedWorkloadVariation",
6                 function = "fixed", value = "50, 100, 200, 400" ],
7              '_Q8jwMEg9Ed2v5eXKEbOQ9g' AS Replication
8               [ strategy = "AbstractActionReplication",
9                 function = "iterate",
10                start = "2", end = "10", step = "2" ]
11 FOR RESOURCE '__oYXADq_EeCCbpF63PfiyA@LAN' AS LAN,
12     RESOURCE '_5uTBUBpmEdyxqpPYxT_m3w@CPU' AS AppServer_CPU,
13     RESOURCE '_tVi40Dq_EeCCbpF63PfiyA@CPU' AS DBServer_CPU,
14     RESOURCE '_tVi40Dq_EeCCbpF63PfiyA@HDD' AS DBServer_HDD,
15     SERVICE  '_u770YEhFEd2v5eXKEbOQ9g' AS svc1
16 USING pcm@'mediastore.properties';
```

Listing 6.5: MediaStore Example for `SELECT` with `EVALUATE DOF`

6.1.5. Summary

The case study in Section 6.1.4 demonstrated that the concepts of DQL are usable to map and implement a scientific approach like the Palladio Bench and PCM. Furthermore users are enabled to conduct performance analyses of PCM instances through a set of queries written in DQL. The functionality of the Palladio Bench could be exploited by means of the Experiment Automation interface together with DQL to offer an experimental environment for simulating PCM model instances with DoFs. A comparable functionality is currently not exposed through the Palladio Bench.

For the Mapping Meta-Model, as integral part of the interfaces between the DQL QEE and the implementation of a DQL Connector, the case study showed that the compromise between a flat-structure for the ease of use and the preservation of architectural information is viable for connecting to established and field-tested solutions for SPE.

An example of the Eclipse IDE showing DQL and PCM integrated to conduct simulations of PCM instances can be seen in Figure 5.15.

6.2. Integrating the KLAPER Approach and KlaperSuite

6.2.1. Introduction to the KLAPER Approach

A general introduction to the KLAPER approach and the *N-to-M Problem* was presented in Section 4.1. KLAPER allows to analyze the behavior of performance model instance through the transformation of design models into performance models. The term *N-to-M Problem* refers in this case to the usage of an intermediate model that allows to define generalized transformations towards the intermediate model from a design meta-model. Outgoing from the intermediate model again generalized transformations can be used towards performance models, that can be analyzed.

By using an intermediate model and by implementing transformations at the meta-model level, realizing support for different software design meta-models (e.g. UML) is a singular effort. After the transformation to the intermediate model is implemented, each valid instance of the meta-model can be used for performance analyses. As the outgoing transformation from the intermediate model towards an analysis model is also a singular effort, users without expert knowledge on specialized mathematical methods and simulation models are able to conduct performance analyses on their software design models

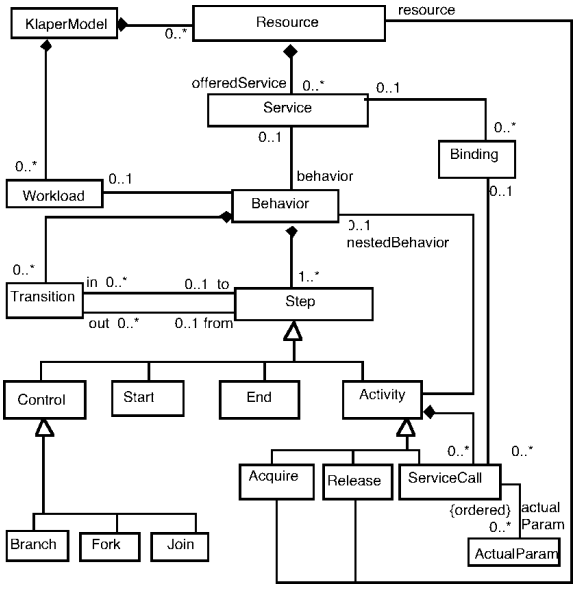


Figure 6.3.: KLAPER Meta-Model from [GMRS08]

[GMRS08, CDF+13]. This section will discuss how KLAPER can be integrated conceptually with the DQL approach.

Following to [CDF+13], KLAPER allows a software architect to start working with KLAPER by two different usage scenarios. Either a software architect might start from a software design model towards KLAPER through model transformation or KLAPER is used KLAPER directly, e.g. if no software design model exists and the creation is not feasible [CDF+13]. As soon as a design model can represented as an instance of the KLAPER intermediate model, it can be transformed into any available analysis model. From the analysis model performance metrics can be derived through using a compatible solving mechanism.

From the perspective of the DQL approach offering a generic user interface through a textual syntax, supporting KLAPER would be a valuable option. On the one hand, the approach of KLAPER is a proven and evaluated scientific approach and on the other hand KLAPER allows to leverage existing transformations (e.g. incoming from UML, outgoing to LQN) for use with DQL. Furthermore the effort for creating new transformations is independent of efforts implementing the Connector [GMRS08]. From a conceptual viewpoint, both approaches could benefit from each other as their objectives are orthogonally.

6.2.2. Mapping KLAPER with the Mapping Meta-Model

The KLAPER Meta-Model, shown in Figure 6.3, is based on the MOF Meta-Model developed by OMG [GMRS08]. The design of the KLAPER Meta-Model is focused on the representation of a software design found in CBSE approaches. The main artifacts found in these designs are entities being either a `Service` and or a `Resource`. From a DQL perspective, these elements can be directly mapped into their corresponding instances of the types `Service` and `Resource` in the Mapping Meta-Model.

Although DoFs are not explicitly supported by KLAPER, the functionality can be emulated. The modeling of DoFs in DQL through modifying elements of the types `Workload` and `ActualParam` in a KLAPER model instance allows to conduct DoF analyses. In [GMRS08] it is suggested to use random variables for `ActualParam` in order to represent dynamics for input parameters during run-time within the performance analysis. DQL

could use instead absolute values and conduct several analyses, which would be similar to the approach for the Palladio Bench in Section 6.1.4.

The remaining elements from the KLAPER Meta-Model are related to modeling the behavior of a software system. For DQL these parameters are out of scope. Summing up, the mapping of KLAPER to the means of DQL is, from a conceptual viewpoint, possible. The central concept of `Service` and `Resource` is likewise for both approaches and the chosen abstraction layer through the Mapping Meta-Model in DQL shows a viable option for a generic mapping with the KLAPER Meta-Model.

6.2.3. Integrating KlaperSuite as Connector

The KlaperSuite is a tool-chain based on the KLAPER intermediate modeling approach [CDF+13]. It provides a set of transformations into the KLAPER intermediate model and from the intermediate model towards analysis models allowing to start performance analyses out of the box. All necessary steps for executing transformations, parameterizing model solvers and executing model solvers are triggered automatically.

The technical foundation of the KlaperSuite implementation is the Eclipse Platform. Model transformations are implemented through QVT, a standardized transformation language used in MDSD [Obj11d]. For extending KlaperSuite to support additional input models, new QVT transformations are necessary to be deployed together with KlaperSuite. Further implementation tasks are not required as the intermediate model serves as an abstraction layer between the incoming instance of the meta-model and the outgoing transformations towards model solvers. For extending the analysis capabilities, developers can supply additional Eclipse Bundles containing solvers. These Bundles need to be deployed in the run-time environment.

Following [CDF+13], the results of an analysis are returned as plain text as the current implementation does not offer an user interface. Using DQL and KlaperSuite as a DQL Connector, this shortcoming could be solved. As the core principles of KLAPER and DQL show similarities, using the Mapping Meta-Model as output format for model solvers in KLAPER could be an approach for integrating KLAPER and DQL. The results obtained through the model solvers, could be accessed through queries in DQL. Furthermore, as both KLAPER and DQL are built up on the Eclipse Platform, the run-time comes at no cost of overhead.

6.2.4. Summary

In Section 6.2.2 the alignment of the core meta-models of KLAPER and DQL were evaluated. The evaluation showed a structural similarity, that allows to integrate the KlaperSuite as a DQL Connector. Section 6.2.3 outlines the motivation for using KlaperSuite as a DQL Connector, allowing to reuse the already existing transformations and benefit from the efforts achieved for automating the analyses of models.

Using DQL as interface to KlaperSuite, which would allow to use the Mapping Meta-Model for storing and representing results and to use DQL for controlling and automating the model analyses, which would address the lacks of the implementation stage of KlaperSuite as described in [CDF+13].

Through integrating KlaperSuite, both approaches would benefit from each other. For DQL, the applicability in the domain of SPE would increase significantly and the development of DQL and KlaperSuite could focus on their core usage scenarios.

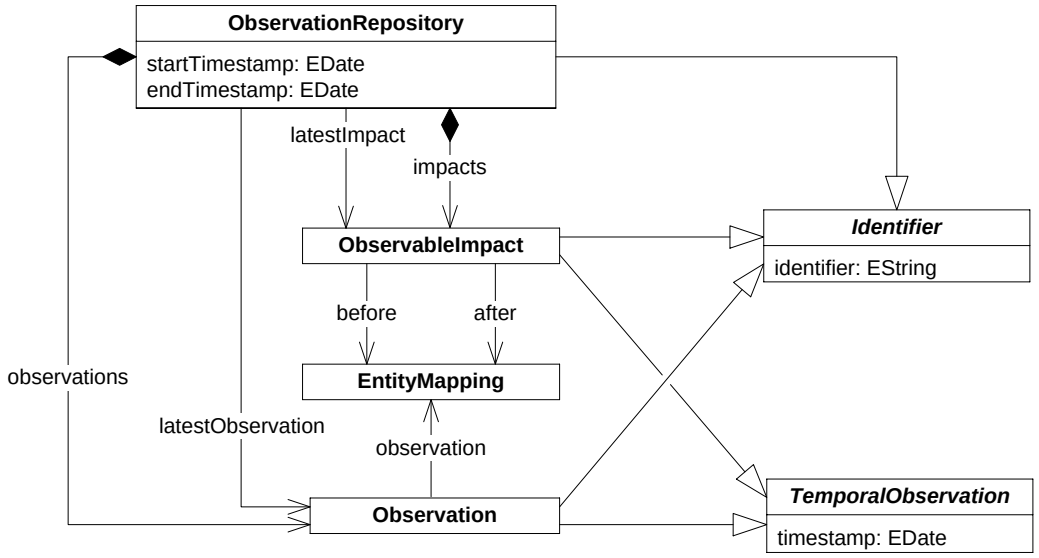


Figure 6.4.: The Repository Meta-Model as Extension of the Mapping Meta-Model

6.3. Implementing a Performance Data Repository (PDR)

6.3.1. Introduction to the Performance Data Repository

In Section 5.2 the Mapping Meta-Model is introduced. The meta-model is used as an abstraction layer from architecture-level performance model to the representation of performance metrics and the identification of model entities. For the evaluation of the expressiveness of the `OBSERVE` expression, a meta-model reusing the Mapping Meta-Model is implemented to offer access to historic performance data. The capabilities of this meta-model can be referred to as an approach of a basic *PDR*.

By the term PDR we refer to a data structure to persist performance metrics with a temporal dimension and a way to link model instances for comparisons. The meta-model introduced in this section is designed to be used for evaluation purposes. Future work may enhance the PDR to support more complex operations and extend the temporal dimension.

Additionally this part of the evaluation is intended to show the reusability of the Mapping Meta-Model. The Mapping Meta-Model is not modified to be embedded into the PDR. But a data structure embedding the Mapping Meta-Model, i.e. the PDR, allows to enhance the Mapping Meta-Model with a temporal dimension. To evaluate the PDR and `OBSERVE` expression, the operations outlined in Section 6.3.3 and Section 6.3.4 are implemented in a DQL Connector; other operations implemented are omitted, but examples are provided with the implementation. The Connector is identified through the model family `pdr` in DQL queries.

6.3.2. Description of the Repository Meta-Model

The basic PDR approach, the Repository Meta-Model, is shown in Figure 6.4. Using `ObservationRepository` the Repository Meta-Model allows to store `Observations` and `ObservableImpacts`. Both are derived from `Identifier`, a stub for identifiers of model entities that should be replaced later, and `TemporalObservation`, a super-type for model entities with a temporal dimension through a time stamp.

An `Observation` is a container for an `EntityMapping` and provides a temporal dimension through inheriting `TemporalObservation`. A way to access historic instances of `EntityMapping` is shown in Section 6.3.3.

`ObservableImpact` is intended to store references to instances of `EntityMapping` as `before` and `after` a point in time in order to provide a view to the difference of these model instances. Therefore an `ObservableImpact` should not directly be referenced on the temporal dimension, but it is referenced by a reference through an `Identifier`. An example is presented in Section 6.3.4.

In the further examples built on top of the Repository Meta-Model, the assumption is made that each model instance has the same structure (i.e. the same entities and probes) as found in the latest model instance. Throughout all model instances, identifiers are assumed to be absolute and unique. Using OCL the Repository Meta-Model could be extended to ensure these invariants.

6.3.3. Accessing historic Model Instances

The query shown in Listing 6.6 is an instance of a `SELECT` query to retrieve three metrics for two different model entities. The keyword `OBSERVE` is used to indicate a ConnectorTimeUnitClause. A ConnectorTimeUnitClause expresses a count of time units and an identifier to refer to a Connector-dependent time unit. The interpretation of count and unit is subject to the DQL Connector. In case of this query, the time unit *before* is referenced with a count of 3.

```
1 SELECT s10.cpuUtilization, s10.netUtilization, w1.queueLength
2 FOR RESOURCE 'srv10' AS s10, SERVICE 'webapp1' AS w1
3 USING pdr@'simple.repository'
4 OBSERVE 3 before;
```

Listing 6.6: Repository Meta-Model Example for `SELECT` with `OBSERVE` requesting a model instance three steps in the past

The implementation of the DQL Connector for PDR interprets the time unit *before* as steps backwards in time starting at the latest `Observation`. For the evaluation it is assumed that the right model instance can be retrieved using an index calculated as `index = models.size() - stepsBefore - 1`. Hence the index computation assumes an temporal order of model instances, which is not assured through OCL in the meta-model.

The ConnectorTimeUnitClause allows to implement Connector-specific time units while being aligned to the DQL and natural language. As the time units are not being specified at the language level, the universality of time expressions can be specialized at a suitable implementation layer. A DQL Connector might delegate the interpretation of time units to a data source, e.g. a RDBMS that serves the Connector with model instances.

6.3.4. Computation of Impacts

Concluding to the support of `Observations`, the Connector offers support to compute the impact of two model instances stored in an `ObservableImpact`. In Listing 6.7 the `OBSERVE` keyword specifies a ConnectorInstanceReferenceClause, that is composed of a Connector-dependent reference name *Impact* and its identifier `impact1` referring directly to an `ObservableImpact` in the model instance `simple.repository` by its identifier.

```
1 SELECT s10.cpuUtilization, s10.netUtilization, w1.queueLength
2 FOR RESOURCE 'srv10' AS s10, SERVICE 'webapp1' AS w1
3 USING pdr@'simple.repository'
4 OBSERVE Impact 'impact1';
```

Listing 6.7: Repository Meta-Model Example for `SELECT` with `OBSERVE` referring to an `ObservableImpact`

To compute metrics from the `ObservableImpact`, the difference of all `DecimalResults` is calculated as subtracting from their preceding value. The impact computation is illustrated by Figure 6.5. Figure 6.5a shows an example for an instance of the `ObservableImpact` with instances of `EntityMapping` attached as `after` and `before`. It is assumed that the structure of the instances of `EntityMapping` is equal, i.e. consisting of the same `Resources`, `Services` and `DecimalResults`. Using this assumption, the DQL Connector can compute the impact of those instances as the difference. The resulting instance of `EntityMapping` which is returned to the user as result is shown in Figure 6.5b.

To sum up, as in the preceding example, the implementation of ConnectorInstanceReferenceClause allows Connector-specific references to express more complex temporal relations being modeled on top of other model elements in the PDR. The motivation for this additional clause is to support the natural language structure, as otherwise the order of unit, i.e. a reference name, and reference itself, i.e. a reference identifier, would contradict to the common language.

6.4. Controlling Experiments with the Software Performance Cockpit (SoPeCo)

6.4.1. Introducing the Software Performance Cockpit

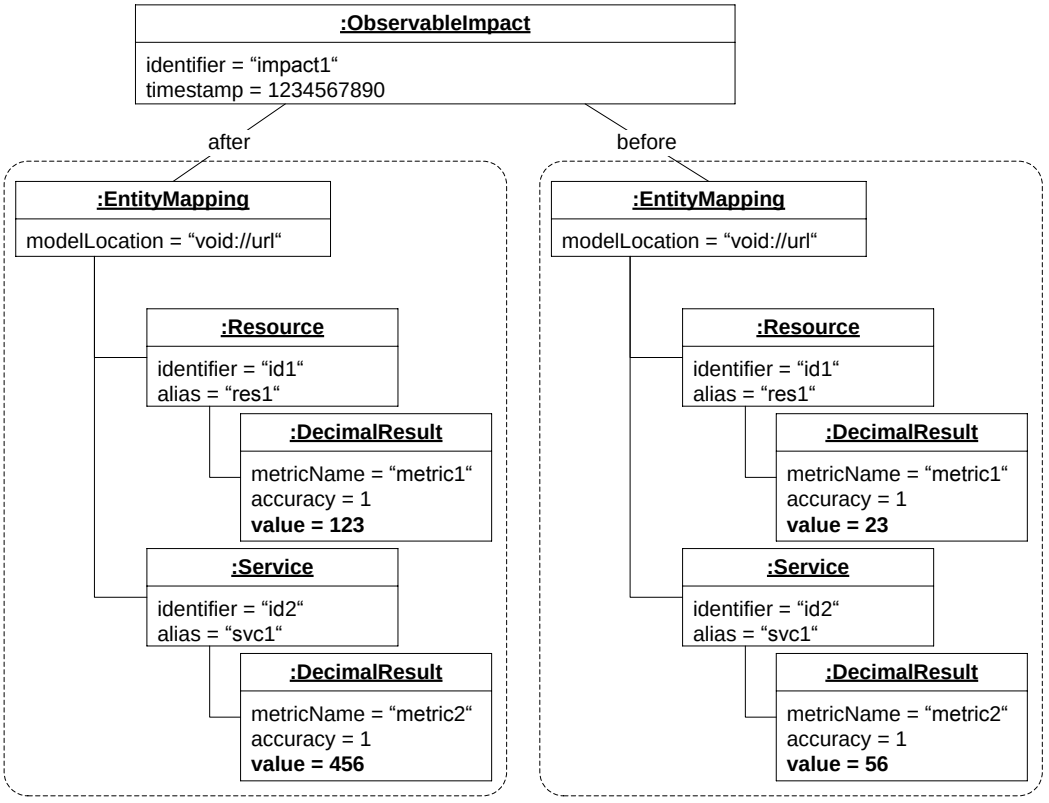
SoPeCo[4] is a framework for the sensitivity analysis of parameter variations in experimental settings. As founded in the SPE-domain, the focus is set on performance model instances, but not limited to. It allows to conduct parameter variation based on statistical models in order to reduce the amount of required experiment runs while covering a comprehensive parameter space. Missing values can be interpolated with different statistical methods. In Section 4.2 foundations are outlined.

The main software components of SoPeCo are the Core, the WebUI and the MEController. The Core component is responsible for the execution infrastructure, controlling experiment execution and persistence. The WebUI serves the web user interface of SoPeCo to configure experiment runs and visualize results. To control an experiment, the MEController component needs to be implemented. It it used to control the System under Test (SuT), defines the available in- and output parameters and is responsible for the life-cycle. More details can be found in [WHHH10] and in the examples[5].

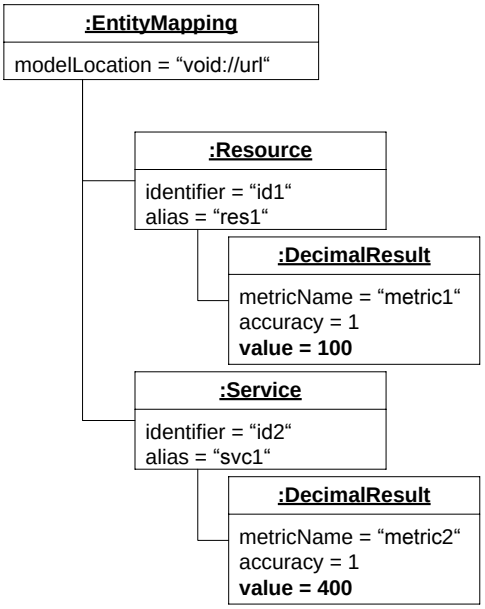
6.4.2. Integrating the Software Performance Cockpit

Although no implementation of the SoPeCo as DQL Connector exists, an approach for a conceptual integration is outlined. The reasons for omitting an implementation in means of this thesis are two-fold: (i) Using SoPeCo requires another component as SuT as the approach is intended solely for systematic parameter variation and not for the actual model solving and (ii) the static architecture of the MEController component is not well-suited to be used with the dynamic query interface of DQL.

[4] `http://www.sopeco.org/home`
[5] `http://www.sopeco.org/tutorials/getting-started/matrix`



(a) Instances of the Mapping Meta-Model aggregated in an `ObservableImpact`



(b) Computed Impact of `ObservableImpact` as Difference between `after` and `before` from 6.5a

Figure 6.5.: Computation of Impacts through `ObservableImpact`

Regarding (i) an example for using SoPeCo in a control loop within the DQL is given in Section 6.4.3. Backgrounds for reasoning (ii) will be explained in the following. The limitations arise from technical issues, while on a conceptual level both approaches could be integrated with.

To implement a MEController, the component offers the abstract superclass `AbstractMEController`. Within this class, the user has to define all input parameters using the annotation `@InputParameter` and all output parameters as `@ObservationParameter`. The SoPeCo run-time analyzes the implementing class for class attributes with these annotations. A user is therefore forced to provide an adequate implementation of the `AbstractMEController` for each model instance to be analyzed.

```
1 SELECT r1.queueLength, s1.responseTime
2 EVALUATE DOF
3          VARYING 'abc123' AS workload
4            [ fuction = "fixed", values = "12, 67, 192"]
5 FOR 'res1' AS r1, 'svc1' AS s1
6 USING solver@'void://url';
```

Listing 6.8: Example of a DoF Query

From a DQL perspective, attributes annotated with `@InputParameter` would be a DoF that can be specified in a `SELECT` query by using the `EVALUATE DOF` keyword. Furthermore using the `VARYING` keyword within `EVALUATE DOF` only a subset of the available DoF can be selected to be varied during experimentation, in Listing 6.8 an example is given. DQL makes therefore no assumptions about available DoF, but allows to put restrictions on DoF or to advice the Connector how to process DoF. In the example a parameter *workload* is varied with a fixed set of values.

To use SoPeCo as DQL Connector for systematic experimentation, for each performance model instance suitable implementations of `AbstractMEController` would be required. Otherwise, users would be limited to analyze only entities of a model instance being explicitly implemented in the implementation of `AbstractMEController`. A way to deal with this issue could be to use functionality of the Java language like the Java Reflection API[6] to modify classes while being executed in a run-time or an alternate implementation of the `AbstractMEController`.

6.4.3. Control Loop for Sensitivity Analysis

To integrate SoPeCo with DQL, given that a solution for the technical limitation described in Section 6.4.1 is found, SoPeCo can be embedded into a DQL Connector and can be used for systematic parameter variation. To use SoPeCo as external tool for parameter variation, the `WITH` keyword within the `EVALUATE DOF` expression is suitable. It can be used to specify the location of an instance of SoPeCo, that shall be queried by the DQL Connector to get parameter setting for DoFs suggested through SoPeCo. Thus the the usage of SoPeCo is subject to the Connector implementation and an instance of SoPeCo can be run independently of the DQL run-time.

The query in Listing 6.9 shows a query from the class of Performance Metric Queries with `EVALUATE DOF` keyword, that should be executed by a DQL Connector for the performance meta-model family `solver`. The `EVALUATE DOF` keyword contains a `WITH` keyword specifying that the evaluation of DoF variation shall be conducted through the `sopeco` handler that can be found at an URL. That way, a Connector being able to solve models,

[6] `http://docs.oracle.com/javase/tutorial/reflect/`

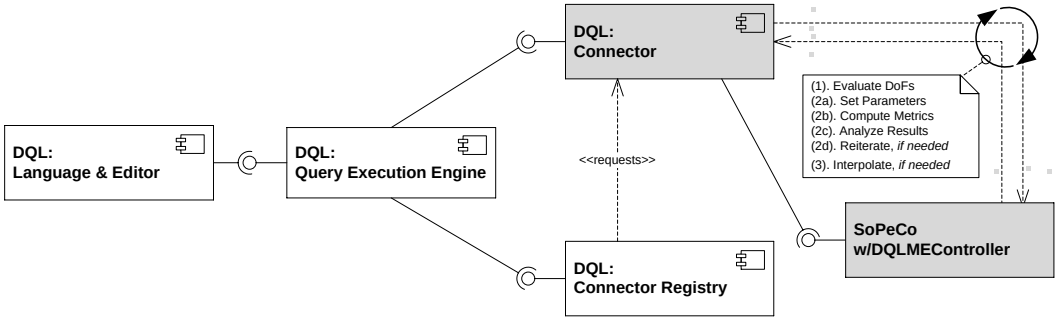


Figure 6.6.: DQL Architecture with a Control Loop with SoPeCo

but unable to explore DoFs, can be extended with the SoPeCo logic consumed as external component.

```
1 SELECT r1.queueLength, s1.responseTime
2 EVALUATE DOF
3         VARYING 'abc123' AS workload
4          [ fuction = "fixed", values = "12, 67, 192"]
5         WITH sopeco@'https://my.host.com/SoPeCo'
6 FOR 'res1' AS r1, 'svc1' AS s1
7 USING solver@'void://url';
```

Listing 6.9: Example of a DoF Query in a Control Loop with SoPeCo

```
1 SELECT r1.queueLength, s1.responseTime
2 FOR 'res1' AS r1, 'svc1' AS s1
3 USING solver@'void://url?param=abc123&value=67';
```

Listing 6.10: Exchange of Queries in the Control Loop with SoPeCo

The `sopeco` handler is used to build a control loop together with the DQL Connector. Figure 6.6 shows the Connector with a component for SoPeCo embedded. In this case, the Connector delegates the exploration of the DoF parameter space to the SoPeCo component. In turn, the SoPeCo component uses the Connector with transformations of the originating model instance to reflect the changes of DoFs and to solve the transformed instances.

For the transformation of model instances for applying DoF parameter settings and to execute queries, a special `DQLMEController` needs to be implemented. The `DQLMEController` instructs the Connector to solve the altered model instances, through reusing the originating query by slicing out parts related to DoFs and with the new model instance to be used. An example is shown in Listing 6.10.

6.4.4. Summary

Using SoPeCo for automating the sensitivity analysis has further advantages for online scenarios. The approach focuses on sophisticated methods for reducing the number of required experiments to calculate metrics through statistical means. Given a query is executed with `CONSTRAINED AS` keyword in the DoFs part, the advice can be delegated

to the SoPeCo instance. SoPeCo can use the advice to interpolate parameter settings of DoFs by wider steps in the parameter space.

Summing up, the output parameters for the sensitivity analysis are defined together with the `SELECT` keyword and the input parameters are either explicitly specified through the `VARYING` keyword or implicitly specified through the model instance. The approach of a control loop allows to integrate both approaches, while leveraging advantages from both. The concept of the `DQLMEController` is reusable across different Connectors by specifying a Connector interface for manipulating model instances — e.g. through an `ALTER` statement in a prospect Model Alteration Query class.

7. Conclusion and Future Work

7.1. Summary and Conclusion

The Descartes Query Language (DQL) approach presented in this thesis shows an option for the integration and unification of different tools for model-based performance predictions. The design of DQL is based on an extensible query classification scheme, which isolates classes from others, thus allowing to extend language parts independently. The implementation is realized using an extensible, component-oriented architecture to provide the necessary means for extending the language functionality incrementally. As the evaluation points out, the applicability of DQL is given and our approach is a viable option for future research activities. Within the evaluation of the language design, exemplary queries are shown and a showcase of a workflow for a performance analysis is presented. As the workflow points out, a performance analysis can be conducted using DQL without the need of the underlying low-level details of the modeling formalism. As users are unburdened from manual efforts and low-level knowledge of the modeling formalism, a main objective of DQL is achieved.

The implementation of DQL is based on the OSGi Framework, which could be leveraged as a suitable foundation for a Component-based Software Engineering (CBSE)-based development approach. Each software component of DQL is enclosed in an OSGi Bundle exposing interfaces, if necessary. To consume functionality of other OSGi Bundles, the OSGi Service Layer is used, which supports the separation of components. A strict separation of components and concerns allows an extensible implementation and changes with foreseeable impact. The interaction between the software components, especially the DQL Query Execution Engine (QEE) as central processing hub, the DQL Connectors providing support for performance models and the DQL Connector Registry as lookup mechanism allows to incorporate support for more performance models. Thus the claim to provide an integrative approach is supported by design and implementation.

The internal Mapping Meta-Model is an important abstraction layer used for data interchange between the software components, for carrying parameterization information and to present results to users. Although the meta-model approach might not yet be complete to represent each available model-based performance prediction approach, the applicability for at least two proven approaches is shown and it might be leveraged for use in future usage scenarios. The evaluation unveiled also that the DQL approach at the current stage is already capable to control an established approach for model-based performance prediction, the Palladio Component Model (PCM) approach. The implemented DQL Connector

is able to control PCM while performing queries for the model structure and to retrieve performance metrics. In case of Degrees of Freedom (DoFs), the DQL Connector allows additionally to perform simulations of PCM instances with varying parameter values, which is not yet part of the PCM simulation engine and user interface.

The evaluation is continued with an outlook to integrate the intermediate model Kernel LAnguage for PErformance and Reliability analysis (KLAPER) as approach for solving the N-to-M transformation problem and Software Performance Cockpit (SoPeCo) to offer a sophisticated solution for efficient experimentation with DoFs. In case of KLAPER and KlaperSuite as its implementation, the evaluation shows that integrating both approaches is viable and results in a win-win-situation for both approaches. DQL addresses shortcomings of the current implementation state, while KLAPER provides means to conduct performance predictions using various descriptive and predictive performance models. For SoPeCo a similar win-win-situation is described conceptually, but technical limitations do not allow yet to integrate both approaches. In both evaluation cases, the evaluation shows conceptual similarities and ways for integrating the approaches.

As final evaluation target, an extension of the Mapping Meta-Model to form a Performance Data Repository (PDR), is introduced. The PDR approach is used to show the capabilities of DQL to extend queries with a temporal dimension without altering the remaining structure of queries. For users this means less effort, while they can focus on their actual query. Another orthogonal feature of DQL, aggregations of performance metrics by statistical means, shows non-invasive extensions while integrating proven approaches. In case of PCM, users can aggregate performance metrics obtained from PCM through DQL queries, which is in this way not supported by PCM but possible through DQL.

The DQL approach as presented by this thesis is completed with an editor provided as plug-in for the Eclipse Integrated Development Environment (IDE). The editor allows users to write queries, execute performance predictions and display results obtained by queries. The editor provides also content assistance to write queries. The content assistance internally uses DQL queries to obtain model-specific information from the underlying DQL Connector to provide advice to the user. Thus using DQL as Application Programming Interface (API) is possible and viable.

Concluding we could achieve our goals and objectives and demonstrate benefits of the DQL approach. Especially as interface for integrating other approaches and unifying interfaces, DQL has its strength. To integrate future extensions, DQL provides the necessary extension points and allows third-party users to integrate their approaches. Extensions can be tailored to provide only necessary parts while reducing efforts for the implementation and benefit from the already existing artifacts. Finally, although the focus of the query language is put on architecture-level performance models with a descriptive nature, the integration of predictive performance models is possible. Hence, the query language allows the integration of conceptual different modeling approaches and provides means to unify different interfaces for different use cases by a single facade. This makes the DQL approach suitable as an integration platform for model-based performance predictions.

7.2. Future Work

During the development of DQL several additional features have been discovered, but are not used as part of the thesis due to time limitations. This section will outline future opportunities and extensions of DQL from a design and implementation viewpoint in addition to the current state of DQL.

Objectives for future work and promising extensions of DQL are:

- **Connector Settings Query Class**: Aligned to the Model Structure Query Class, users should be able to extract information regarding Connector-specific settings through the DQL. Examples for these settings are the currently set exploration strategy or available constraints.
- **Entity Parameterization Query Class**: Besides of extracting performance metrics, the DQL approach could also be used to parameterize performance model instances with data obtained from monitoring systems through additional DQL Connectors.
- **Unification of Query Results from different Sources**: In complex enterprise scenarios multiple information sources exist. An Information Technology (IT) system might be monitored by a set of monitoring systems. In order to assist a performance analyst, DQL could provide a `UNION` expression that allows to combine results from multiple queries into a single result set. The functionality of `UNION` would correspond to its Structured Query Language (SQL) counterpart and could be implemented by merging multiple instances of the Mapping Meta-Model.
- **Filtering Expressions and DQL QEE Support**: In order to increase the readability of results from queries especially in the Model Structure Query Class, that tend to consist of a high amount of result values, filtering expressions and functionality is required. The filter specification should be defined by DQL expressions, interpreted at the DQL QEE and DQL Connector layers and being represented through the Mapping Meta-Model.
- **DQL Connector for VMware vSphere**: VMware vSphere[1] is used in a research project in the Descartes Research Group. To access performance metrics through the monitoring components of vSphere, an API exists. As the API consists of complex and generic classes, the effort for implementing direct API access in custom applications can be considered high. Embedding vSphere through a DQL Connector would simplify working with the API in context of Software Performance Engineering (SPE) research.
- **DQL Connector for Monitoring Tools**: As usage scenario outside of the field of SPE, DQL Connectors for different monitoring systems found in industry could be realized. Users could access different monitoring systems through DQL queries and have a more general view on their systems. Together with the implementation of `UNION` expression different sources for monitoring data could be integrated by DQL as middleware. Candidates for implementations of the DQL Connector could be Nagios[2] as Open Source product or IBM Tivoli Monitoring[3] as commercial product.

Design and the implementation of the DQL approach are realized by using extensible technologies and, supported by a strict separation of concerns, the impact and effort of additional features is quantifiable for new features. Thus, when new features for inclusion in DQL are discovered, their integration is possible while the effort and impact are foreseeable.

[1] `http://www.vmware.com/products/datacenter-virtualization/vsphere/`
[2] `http://www.nagios.org/`
[3] `http://www.ibm.com/software/products/us/en/tivomoni/`

Bibliography

[BDIS04] S. Balsamo, A. Di Marco, P. Inverardi, and M. Simeoni, "Model-Based Performance Prediction in Software Development: A Survey," *IEEE Transactions on Software Engineering*, vol. 30, no. 5, 2004.

[BHK12] F. Brosig, N. Huber, and S. Kounev, "The Descartes Meta-Model," Karlsruhe Institute of Technology, Karlsruhe, Tech. Rep., 2012. [Online]. Available: http://sdqweb.ipd.kit.edu/publications/descartes-pdfs/DMM-TechReport-0.81.pdf

[BKK09] F. Brosig, S. Kounev, and K. Krogmann, "Automated extraction of palladio component models from running enterprise Java applications," in *Proceedings of the 4th International ICST Conference on Performance Evaluation Methodologies and Tools*. ICST, Oct. 2009, p. 10. [Online]. Available: http://dl.acm.org/citation.cfm?id=1698822.1698835

[BKR09] S. Becker, H. Koziolek, and R. Reussner, "The Palladio component model for model-driven performance prediction," *Journal of Systems and Software*, vol. 82, no. 1, pp. 3–22, Jan. 2009. [Online]. Available: http://dx.doi.org/10.1016/j.jss.2008.03.066

[CB76] D. D. Chamberlin and R. F. Boyce, "SEQUEL," in *Proceedings of the 1976 ACM SIGFIDET (now SIGMOD) workshop on Data description, access and control - FIDET '76*. New York, New York, USA: ACM Press, May 1976, pp. 249–264. [Online]. Available: http://dl.acm.org/citation.cfm?id=800296.811515

[CDF+13] 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*, pp. 1–22, Mar. 2013. [Online]. Available: http://link.springer.com/article/10.1007/s10270-013-0334-8/fulltext.html

[Coh10] M. Cohn, *Succeeding with Agile: Software Development using Scrum*, ser. The Addison-Wesley signature series. Upper Saddle River, NJ: Addison-Wesley, 2010. [Online]. Available: http://swb.bsz-bw.de/DB=2.1/PPNSET?PPN=31415941X

[DB78] P. Denning and J. Buzen, "The Operational Analysis of Queueing Network Models," *ACM Computing Surveys (CSUR)*, vol. 10, no. 3, 1978. [Online]. Available: http://dl.acm.org/citation.cfm?id=356735

[DeM09] L. DeMichiel, "JSR 317: Java Persistence 2.0," Java Community Process, Tech. Rep., 2009. [Online]. Available: http://jcp.org/en/jsr/detail?id=317

[Ecl12] Eclipse Foundation, "Xtext 2.3 Documentation," Tech. Rep., 2012. [Online]. Available: http://www.eclipse.org/Xtext/documentation/2.3.0/Documentation.pdf

[FJKH12] S. Frey, R. Jung, B. Kiel, and W. Hasselbring, "MAMBA: Model-Based Software Analysis Utilizing OMG's SMM," no. May, 2012. [Online]. Available: http://eprints.uni-kiel.de/15393/

[Fow10] M. Fowler, *Domain-specific Languages*, ser. Addison-Wesley signature series. Upper Saddle River, N.J.: Addison-Wesley, 2010. [Online]. Available: http://swb.bsz-bw.de/DB=2.1/PPNSET?PPN=355396769

[FPW+06] G. Franks, D. Petriu, M. Woodside, J. Xu, and P. Tregunno, "Layered Bottlenecks and Their Mitigation," in *Third International Conference on the Quantitative Evaluation of Systems - (QEST'06)*. IEEE, 2006, pp. 103–114. [Online]. Available: http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=1703994

[FvHJ+11] S. Frey, A. van Hoorn, R. Jung, W. Hasselbring, and B. Kiel, "MAMBA: A measurement architecture for model-based analysis," no. 1112, 2011. [Online]. Available: http://eprints.uni-kiel.de/14421/

[GJS+11] J. Gosling, B. Joy, G. Steele, G. Bracha, and A. Buckley, "The Java Language Specification," 2011. [Online]. Available: http://docs.oracle.com/javase/specs/

[GMRS08] V. Grassi, R. Mirandola, E. Randazzo, and A. Sabetta, "KLAPER: An Intermediate Language for Model-Driven Predictive Analysis of Performance and Reliability," in *The Common Component Modeling Example*, ser. Lecture Notes in Computer Science, A. Rausch, R. Reussner, R. Mirandola, and F. Plášil, Eds. Springer Berlin / Heidelberg, 2008, vol. 5153, pp. 327–356. [Online]. Available: http://dx.doi.org/10.1007/978-3-540-85289-6_13

[GMS06] V. Grassi, R. Mirandola, and A. Sabetta, "A Model Transformation Approach for the Early Performance and Reliability Analysis of Component-Based Systems," in *Component-Based Software Engineering*, ser. Lecture Notes in Computer Science, I. Gorton, G. Heineman, I. Crnković, H. Schmidt, J. Stafford, C. Szyperski, and K. Wallnau, Eds. Springer Berlin Heidelberg, 2006, vol. 4063, pp. 270–284. [Online]. Available: http://dx.doi.org/10.1007/11783565_19

[HBK12] N. Huber, F. Brosig, and S. Kounev, "Modeling dynamic virtualized resource landscapes," *Proceedings of the 8th international ACM SIGSOFT conference on Quality of Software Architectures - QoSA '12*, p. 81, 2012. [Online]. Available: http://dl.acm.org/citation.cfm?doid=2304696.2304711

[HBR+10] N. Huber, S. Becker, C. Rathfelder, J. Schweflinghaus, and R. H. Reussner, "Performance modeling in industry," in *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering - ICSE '10*, vol. 2. New York, New York, USA: ACM Press, May 2010, p. 1. [Online]. Available: http://dl.acm.org/citation.cfm?id=1810295.1810297

[ILFW05] T. A. Israr, D. H. Lau, G. Franks, and M. Woodside, "Automatic generation of layered queuing software performance models from commonly available traces," in *Proceedings of the 5th international workshop on Software and performance - WOSP '05*. New York, New York, USA: ACM Press, Jul. 2005, pp. 147–158. [Online]. Available: http://dl.acm.org/citation.cfm?id=1071021.1071037

[IPD12] IPD Reussner, "PCM 3.3 - Example Workspace," 2012. [Online]. Available: http://sdqweb.ipd.kit.edu/wiki/PCM_3.3/Example_Workspace

[KB06] S. Kounev and A. Buchmann, "SimQPN - A tool and methodology for analyzing queueing Petri net models by means of simulation," *Performance Evaluation*, vol. 63, no. 4-5, pp. 364–394, May 2006. [Online]. Available: http://linkinghub.elsevier.com/retrieve/pii/S0166531605000477

[KBHR10] S. Kounev, F. Brosig, N. Huber, and R. Reussner, "Towards Self-Aware Performance and Resource Management in Modern Service-Oriented Systems," in *2010 IEEE International Conference on Services Computing*. IEEE, Jul. 2010, pp. 621–624. [Online]. Available: http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=5557287

[Kou05] S. Kounev, "Performance Engineering of Distributed Component-Based Systems- Benchmarking, Modeling and Performance Prediction," Ph.D. dissertation, Darmstadt University of Technology, Aug. 2005.

[Kou06] ——, "Performance Modeling and Evaluation of Distributed Component-Based Systems Using Queueing Petri Nets," *IEEE Transactions on Software Engineering*, vol. 32, no. 7, pp. 486–502, 2006.

[Koz10] H. Koziolek, "Performance evaluation of component-based software systems: A survey," *Performance Evaluation*, vol. 67, no. 8, pp. 634–658, Aug. 2010. [Online]. Available: http://linkinghub.elsevier.com/retrieve/pii/S016653160900100X

[KR11] A. Koziolek and R. Reussner, "Towards a generic quality optimisation framework for component-based system models," *Proceedings of the 14th international ACM Sigsoft symposium on Component based software engineering - CBSE '11*, p. 103, 2011. [Online]. Available: http://portal.acm.org/citation.cfm?doid=2000229.2000244

[KSM11] S. Kounev, S. Spinner, and P. Meier, "Introduction to Queueing Petri Nets: Modeling Formalism, Tool Support and Case Studies," *Proceedings of the 3rd ACM/SPEC International Conference on Performance Engineering (ICPE 2012), Boston, USA, 22-25 April 2012*, pp. 9–18, 2011.

[LCW+09] J. Li, J. Chinneck, M. Woodside, M. Litoiu, and G. Iszlai, "Performance model driven QoS guarantees and optimization in clouds," in *2009 ICSE Workshop on Software Engineering Challenges of Cloud Computing*. IEEE, 2009, pp. 15–22. [Online]. Available: http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=5071528

[MAD94] D. A. Menasce, V. A. F. Almeida, and L. W. Dowdy, *Capacity Planning and Performance Modeling˜- From Mainframes to Client-Server Systems*. Prentice Hall, Englewood Cliffs, NG, 1994.

[MDA04] D. A. Menasce, L. W. Dowdy, and V. A. F. Almeida, *Performance by Design: Computer Capacity Planning By Example*. Upper Saddle River, NJ, USA: Prentice Hall PTR, Jan. 2004. [Online]. Available: http://dl.acm.org/citation.cfm?id=995/032

[Mer11] P. Merkle, "Comparing Process- and Event-oriented Software Performance Simulation," Master Thesis, Karlsruhe Institute of Technology, 2011. [Online]. Available: http://sdqweb.ipd.kit.edu/publications/pdfs/merkle2011a.pdf

[Obj05] Object Management Group, "UML Profile for Schedulability, Performance, and Time Specification 1.1," no. January, 2005. [Online]. Available: http://www.omg.org/spec/SPTP/1.1/

[Obj11a] ——, “Meta Object Facility Core Specification 2.4.1,” 2011. [Online]. Available: http://www.omg.org/spec/MOF/2.4.1/

[Obj11b] ——, “Meta Object Facility (MOF) 2.0 Query/View/Transformation,” 2011. [Online]. Available: http://www.omg.org/spec/QVT/1.1/

[Obj11c] ——, “Unified Modeling Language 2.4.1 - Infrastructure,” 2011. [Online]. Available: http://www.omg.org/spec/UML/2.4.1/

[Obj11d] ——, “Unified Modeling Language 2.4.1 - Superstructure,” 2011. [Online]. Available: http://www.omg.org/spec/UML/2.4.1/

[Obj12] ——, “Structured Metrics Metamodel 1.0,” 2012. [Online]. Available: http://www.omg.org/spec/SMM/1.0/

[OSG11] OSGi Alliance, “OSGi Service Platform Core Specification - Release 4, Version 4.3,” no. April, 2011. [Online]. Available: http://www.osgi.org/Download/Release4V43

[OSG12] ——, “OSGi Service Platform Service Compendium - Release 4, Version 4.3,” no. January, 2012. [Online]. Available: http://www.osgi.org/Download/Release4V43

[RBB+11] R. Reussner, S. Becker, E. Burger, J. Happe, M. Hauck, A. Koziolek, H. Koziolek, K. Krogmann, and M. Kuperberg, “The Palladio Component Model,” Chair for Software Design & Quality, Faculty of Informatics, Karlsruhe Institute of Technology, Karlsruhe, Tech. Rep. March, 2011. [Online]. Available: http://digbib.ubka.uni-karlsruhe.de/volltexte/1000022503

[RBKR12] 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 *2012 Joint Working IEEE/IFIP Conference on Software Architecture and European Conference on Software Architecture*. IEEE, Aug. 2012, pp. 31–40. [Online]. Available: http://ieeexplore.ieee.org/xpl/articleDetails.jsp?arnumber=6337759

[Sak09] K. Saks, “JSR 318: Enterprise JavaBeans 3.1,” Java Community Process, Tech. Rep., 2009. [Online]. Available: http://jcp.org/en/jsr/detail?id=318

[SBPM09] D. Steinberg, F. Budinsky, M. Paternostro, and E. Merks, *EMF: Eclipse Modeling Framework*. Addison-Wesley, 2009. [Online]. Available: http://books.google.com/books?id=oAYcAAAACAAJ&pgis=1

[SKM12] S. Spinner, S. Kounev, and P. Meier, “Stochastic Modeling and Analysis using QPME: Queueing Petri Net Modeling Environment v2.0,” in *Proceedings of the 33rd International Conference on Application and Theory of Petri Nets and Concurrency (Petri Nets 2012)*, ser. Lecture Notes in Computer Science (LNCS), S. Haddad and L. Pomello, Eds., vol. 7347. Springer-Verlag, 2012, pp. 388–397.

[SL11] C. Smith and C. Lladó, “Model Interoperability for Performance Engineering: Survey of Milestones and Evolution,” in *Performance Evaluation of Computer and Communication Systems. Milestones and Future Challenges*, ser. Lecture Notes in Computer Science, K. Hummel, H. Hlavacs, and W. Gansterer, Eds. Springer Berlin Heidelberg, 2011, vol. 6821, pp. 10–23. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-25575-5_2

[SLP10] 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*, vol. 67, no. 7, pp. 548–568, Jul. 2010. [Online]. Available: http://dx.doi.org/10.1016/j.peva.2010.01.006http://linkinghub.elsevier.com/retrieve/pii/S0166531610000076

[SW02] C. U. Smith and L. G. Williams, *Performance Solutions - A Practical Guide to Creating Responsive, Scalable Software*. Addison-Wesley, 2002. [Online]. Available: dl.acm.org/citation.cfm?id=500664

[TNG05] E. Thereska, D. Narayanan, and G. Ganger, "Towards Self-Predicting Systems: What If You Could Ask "What-If"?" in *16th International Workshop on Database and Expert Systems Applications (DEXA'05)*. IEEE, 2005, pp. 196–200. [Online]. Available: http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=1508272

[WFP07] M. Woodside, G. Franks, and D. C. Petriu, "The Future of Software Performance Engineering," in *Future of Software Engineering (FOSE '07)*. IEEE, May 2007, pp. 171–187. [Online]. Available: http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=4221619

[WHHH10] D. Westermann, J. Happe, M. Hauck, and C. Heupel, "The Performance Cockpit Approach: A Framework For Systematic Performance Evaluations," in *2010 36th EUROMICRO Conference on Software Engineering and Advanced Applications*. IEEE, Sep. 2010, pp. 31–38. [Online]. Available: http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=5598076

[WHKF12] D. Westermann, J. Happe, R. Krebs, and R. Farahbod, "Automated inference of goal-oriented performance prediction functions," in *Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering - ASE 2012*, ser. ASE 2012. New York, New York, USA: ACM Press, 2012, p. 190. [Online]. Available: http://doi.acm.org/10.1145/2351676.2351703http://dl.acm.org/citation.cfm?doid=2351676.2351703

[WHSB01] M. Woodside, C. Hrischuk, B. Selic, and S. Bayarov, "Automated performance modeling of software generated by a design environment," *Performance Evaluation*, vol. 45, no. 2-3, pp. 107–123, Jul. 2001. [Online]. Available: http://dx.doi.org/10.1016/S0166-5316(01)00033-5

[WKH11] D. Westermann, R. Krebs, and J. Happe, "Efficient Experiment Selection in Automated Software Performance Evaluations," in *Computer Performance Engineering*, ser. Lecture Notes in Computer Science, N. Thomas, Ed. Springer Berlin Heidelberg, 2011, vol. 6977, pp. 325–339. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-24749-1_24

[WPP+05] M. Woodside, D. C. Petriu, D. B. Petriu, H. Shen, T. Israr, and J. Merseguer, "Performance by unified model analysis (PUMA)," in *Proceedings of the 5th international workshop on Software and performance - WOSP '05*. New York, New York, USA: ACM Press, Jul. 2005, pp. 1–12. [Online]. Available: http://dl.acm.org/citation.cfm?id=1071021.1071022

A. Acronyms

ADM Architecture Driven Modernization.

API Application Programming Interface.

CBSE Component-based Software Engineering.

CI Continuous Integration.

CSM Core Scenario Model.

DMM Descartes Meta-Model.

DoF Degree of Freedom.

DQL Descartes Query Language.

DSL Domain-specific Language.

EBNF Extended Backus-Naur Form.

EJB Enterprise JavaBean.

EMF Eclipse Modeling Framework.

IDE Integrated Development Environment.

IT Information Technology.

JPA Java Persistence API.

KLAPER Kernel LAnguage for PErformance and Reliability analysis.

LQN Layered Queueing Network.

MAMBA Measurement Architecture for Model-Based Analysis.

MDA Model-driven Architecture.

MDSD Model-driven Software Development.

MOF Meta Object Facility.

MQL MAMBA Query Language.

OCL Object Constraint Language.

OMG Objects Management Group.

PCM Palladio Component Model.

PDR Performance Data Repository.

PMIF Performance Model Interchange Format.

PN Petri Net.

PUMA Performance by Unified Model Analysis.

QEE Query Execution Engine.

QN Queueing Network.

QoS Quality of Service.

QPN Queueing Petri Network.

RAM Random Access Memory.

RDBMS Relational Database Management System.

RDSEFF Resource Demanding Service Effect Specification.

SCR Service Component Registry.

SE Software Engineering.

SEFF Service Effect Specification.

SEQUEL Structured English Query Language.

SLA Service Level Agreement.

SMM Structured Metrics Metamodel.

SoPeCo Software Performance Cockpit.

SPE Software Performance Engineering.

SQL Structured Query Language.

SuT System under Test.

SVN Subversion.

UI User Interface.

UML Unified Modeling Language.

UML-SPT UML Profile for Schedulability, Performance and Time.

URI Uniform Resource Identifier.

UUID Universally Unique Identifier.

vCPU virtual Central Processing Unit.

VM Virtual Machine.

XML Extensible Markup Language.

B. Index of DQL Grammar Rules

C. Implemented Software Components

The following software components are implemented as part of this thesis. The Subversion (SVN) Repository Root containing all source codes can be found at `https://svnserver.informatik.kit.edu/i43/svn/descartes/code/querylang/`.

- **DQL Connector**
  - `edu.kit.ipd.descartes.ql.connector`: Interfaces for implementing a Connector
  - `edu.kit.ipd.descartes.ql.connector.nop`: NOP (No Operation) Connector, for testing purposes
  - `edu.kit.ipd.descartes.ql.connector.pcm`: PCM (Palladio Component Model) Connector
  - `edu.kit.ipd.descartes.ql.connector.pdr`: PDR (Performance Data Repository) Connector
- **DQL Language & Editor**
  - `edu.kit.ipd.descartes.ql.lang`: Main Xtext project
  - `edu.kit.ipd.descartes.ql.lang.sdk`: Xtext generated contents
  - `edu.kit.ipd.descartes.ql.lang.ui`: Xtext generated contents
- **DQL Query Execution Engine (QEE)**
  - `edu.kit.ipd.descartes.ql.core.engine`: Query Execution Engine (QEE)
  - `edu.kit.ipd.descartes.ql.core.platform.eclipse`: Platform-specific Code for Eclipse
  - `edu.kit.ipd.descartes.ql.core.platform.external`: Platform-specific Code for integrating Generic Java/OSGi Applications
  - `edu.kit.ipd.descartes.ql.core.registry`: Connector Registry
- **Models**
  - `edu.kit.ipd.descartes.ql.models.mapping`: Mapping Meta-Model, Eclipse Modeling Framework (EMF)-based
  - `edu.kit.ipd.descartes.ql.models.repository`: Performance Data Repository (PDR), EMF-based