Declarative Automation Framework for QPME based on DQL

Bachelor Thesis of

Simon Eismann

At the Department of Computer Science
Chair for Computer Science II
Software Engineering

Reviewer:	Samuel Kounev
Advisor:	Dipl Inform Jürgen Walter
Second advisor:	Msc. Simon Spinner

Duration: 12. January 2015 – 24. March 2015

Contents

1. Introduction

Performance engineering dates back to 1909 when Agner Krarup Erlang was given the task to analyze the performance of telephone networks [PR10]. He showed that queuing theory could be used to predict the required number of circuits for good telephone coverage. This method also proved effective to analyze the performance of software systems [BDMIS04], during the software's design phase as well as to run time. It can be used to analyze the performance of a system hosted on a single server as well as for distributed component-based systems. When analyzing the performance of a software system questions like the following arise::

- How long does a user have to wait on average until his request is handled?
- What is the longest time a user has to wait?
- How many database/application servers are required to fulfill a certain service level agreement (SLA)??
- What is the highest number of concurrent users the system can handle?

To analyze a system, predict its future system states and answer questions like the previously mentioned the system has to be modeled. Since the beginning of performance engineering, the area has evolved. There exists a variety of modeling formalisms. Predictive models, like, for example, Queueing Nets (QNs), Layer QNs, Queueing Petri Nets (QPNs). Architectural performance models, like Descartes Modeling Language (DML) or Palladio Component Model (PCM) can be transformed to predictive models for analysis. The model can be analyzed using analytical or simulation approaches (see 2.3) provided by formalism specific tooling. Modeling formalisms and their corresponding tooling have different advantages and disadvantages. Performing an educated choice of formalism and tooling is difficult. This means engineers have to understand multiple formalisms, know their strengths and know how to use the corresponding tooling. An attempt to simplify this is the Descartes Query Language (DQL) [GBK14], a query based language that utilizes an adapter based approach to support a multitude of modeling formalisms. It enables the user to specify what performance questions and metrics he is interested in. DQL requires a model of the system in a supported modeling formalism, but to use DQL the user does not need to know or understand the used modeling formalim. Using DQL the user can retrieve information about the model, the values of performance metrics and even use automated detection of performance issues. DQL also allows the user to run multiple simulation on a model while automatically changing some details about the model. This approach brings multiple advantages:

- The user does not need to understand the underlying modeling formalism to use DQL. This means one person can model the system using his preferred formalism and others can analyze the system via DQL without having any understanding of the used modeling formalism.
- Utilizing automated model transformation the user can transform the model to a formalism that fits his needs and run DQL queries on it without having to learn about a second formalism or its tooling. For example a formalism specialized on design time analysis might be used during the development of a product and the model is later automatically transformed to a formalism for run time analysis.
- DQL filters the collected data and only returns the requested information, making the results more lucid. As mentioned above an user usually has a specific in question in mind during a performance analysis, which he formulate as a DQL query. DQL then answers the users question.
- The option to run multiple simulation on a model while automatically changing some details about the model makes it easy to for example analyze if/how much the performance of a system would improve if it had three, five or seven database servers.

To support a variety of modeling formalisms DQL uses a modularized approach, it delegates the actual analysis to a formalism specific component called connector. This allows a plugin style development. While the DQL engine takes care of the calculation of aggregates (sums, averages, ...), the connector can freely decide how he handles most matters. The connector chooses which entities with what metrics it offers, or what model properties can automatically be varied during batch simulation. The connector can also decides which values it allows for the runtime guiding `CONSTRAINED AS` clause. This freedom makes the implementation of a connector to more than a simple implementation task, a lot design choices have to be made. All currently implemented connectors support architecture level performance models, like PCM [BHK11]. But with the amount of freedom the connectors have it should also be possible to use DQL for a run time analysis modeling formalism. In this thesis we want to test this hypothesis by implementing a connector for simQPN, a simulator for QPNs. QPNs are a modeling formalism that was published first by Bause in 1993 [Bau93]. They consist of a combination of two prior modeling formalisms, the Petri Nets (PNs) and QNs. The result is a modeling formalism that hardware contention, scheduling strategies, simultaneous resource possession, synchronization, blocking and software contention can be modeled with [KB06]. While many analysis methods for QNs can also be used on QPNs, simQPN utilizes simulations to analyze QPNs since this approach scales better [KB06]. We chose QPNs over other modeling formalims, since for QPNs there already exists extensive tooling, for example Queueing Petri net Modeling Environments (QPMEs) which also uses simQPN as a simulator. Aside from proving that DQL has uses outside of analyzing architecture level models, this will also give users of simQPN access to the advantages of DQL. For new users configuring simQPN simulations can be quite confusing, while the proposed connector will always guarantie that a valid configuration is used. This means the user can start simulations without having any knowledge about how simulations work. As mentioned above, an user usually has a very specific question in mind when starting a simulation, so DQLs query - answer style leads to lucid results. Also simQPN currently does not support the batch analysis of slightly different models, so the connector will also provide new features for the users of QPN.
chapter 2 introduces the foundations needed to understand this thesis, this includes DQL, QPNs, some basics for the simulation of the latter and a short introduction to QPME. Then in section 3.2 we explain our approach for the design and implementation of a simQPN connector for DQL. It covers our goals, the plan for the implementation and evaluation of the connector. Before we could start on the implementation of the connector

we had to make make a series of design choices. Which QPN entities should we use as DQL entities? How should the `CONSTRAINED AS` feature be implemented? What properties of the model can the user variate during batch execution? These issues among others are discussed in chapter 4. The technical details for the implementation of the three subconnectors, the Model Structure Query Connector, the Performance Metric Query Connector and the Performance Issue Query Connectorcan be found in chapter 5. Afterwards we evaluate the connector in chapter 6, this includes unit testing as well as a performance test. The concluding chapter 7 summarizes the thesis and lists ideas for future work.

2. Foundations

This chapter explains the foundations needed to understand this thesis. In section 2.1 the main subject of this thesis DQL will be explained. Then the structure of QPNs is shown in section 2.2, followed by details about the simulation of the latter in section 2.3. Lastly QPME, a tool for modelling and analysis of QPNs will be introduced in section 2.4. QPME consists of two independent tools, Queueing Petri net Editor (QPE) and simQPN. QPE is a graphical interface for the creation and manipulation of QPNs. SimQPN is QPMEs simulator for QPNs.

2.1 Descartes Query Language

The Descartes Query Language (DQL) provides a common interface for various performance modeling formalisms. This abstraction level enables the user to access multiple modeling formalisms through one interface. The user can use queries to select which performance metrics should be calculated for the system. Performance metrics are values which to indicate the performance of the system. An example for a performance metric would be the average time an user has to wait for completion of a request. To use DQL to analyze a network, it first has to be described using one of the supported modeling formalisms, like PCM. The syntax of DQL is similar to the Structured Query Language (SQL), which makes it easy to understand.
The query language DQL supports three types of queries:

- Model Structure Queries
- Performance Metrics Queries
- Performance Issue Queries

Model Structure Queries are the basic queries that every analysis starts with. They can be identified by the `LIST` keyword, which every Model Structure Query starts with. The analysis of a system usually starts of with a query to `LIST` all available entities. Entities represent every instance over which statistics can be compiled. Next comes a query to `LIST` all available performance metrics for the entities of interest. If the user plans on using Degree of Freedom (DoF), he can also formulate a Model Structure Query to `LIST` the available DoF options. The core of DQL are the so-called *Performance Metric Queries.* These queries specify which information about the systems performance is required. In Table 2.1 you can see three examples for DQL queries. The first one is a basic query to

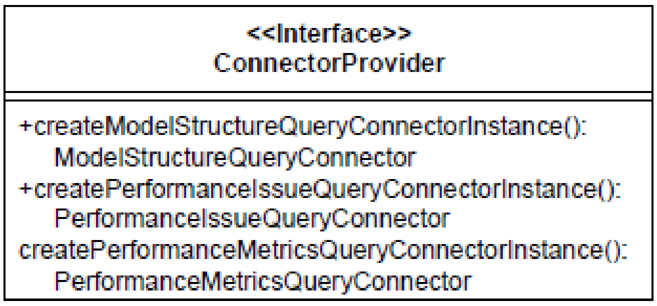
Querytype	**Query**	**Result**
Model Structure	`LIST ENTITIES` `USING` connector@'modelpath';	`RESOURCE` Entity1 `RESOURCE` Entity2
Model Structure	`LIST METRICS` (`RESOURCE` 'Entity1' `AS` Client, `RESOURCE` 'Entity12' `AS` Database) `USING` connector@'modelpath';	Entity1.metric1 Entity1.metric2 Entity2.metric3 Entity2.metric4
Performance Metric	`SELECT` Client.metric1, Database.metric4 `CONSTRAINED AS` 'Accurate' `FOR` `RESOURCE` 'Entity1' `AS` Client, `RESOURCE` 'Entity2' `AS` Database `USING` connector@'modelpath';	Client.metric1: 0.3 Database.metric4: 7

Table 2.1: Example DQL-Queries with Results

`LIST` all available entities, which reveals that there are two entities, `Entity1` and `Entity2`. With the next query the user wants to know which metrics are available for both entities. The query returns four available metrics, the `metrics1-4`. With this knowledge the user now decides that he is interested in `metric1` for `Entity1` and `metric4` for `Entity2`. He also uses the `CONSTRAINED AS` feature to specify that he prefers accurate results over execution speed. In the second and third query, DQL's aliasing feature was used to rename the non descriptive `Entity1/Entity2` to `Database` and `Client` respectively. The last line of every DQL query contains the `USING` clause that specifies which connector should be used to obtain the desired metrics and the path to the model that should be analyzed. Aside from the Model Structure and Performance Metrice Queries there are the *Performance Issue Queries*. As the name suggests these queries can be used to find performance issues. They currently only support the option to `DETECT BOTTLENECKS`, but will be expanded in the future. It should be noted, that to date the `DETECT BOTTLENECK` feature is not established for any modeling formalism. The Performance Issue Queries are a step towards the descartes vision of self-aware computing. A self-aware computer is given a goal and optimizes it's own execution in order to achieve this goal with the minimal amount of resources and energy [AME^{+}09]. In theory a program could use Performance Issue Queries to analyze which parts of the system are bottlenecks for it's performance and then optimize it's own structure accordingly if it expects load spikes.
DQL utilizes a modularized approach in order to support multiple modeling formalisms. Every incoming query is first fed into to Query Execution Engine (QEE), which validates the query and transforms the query from a string to the internal query format. The QEE then requests the connector for the used modeling formalism from the connector registry. This connector then compiles the requested information and sends them back to the QEE, which then validates the results, calculates aggregates if necessary and displays the results to the user. Every DQL connector consists of the following pieces:

- OSGi Bundle Activator
- Connector Provider
- Connectors for the supported Query types

DQL implements a architecture defined by Open Services Gateway Initiative (OSGi)), a approach for fully modulized programs. It allows you to install, uninstall, start and stop components without having to restart the whole program. The OSGi Bundle Activator is part of the OSGi Life cycle for a component and is called during the starting period of the the application. It initializes the OSGi bundle context and any other resources that might be required. The Connector Provider is a interface in DQL, that implements the functions shown in Figure 2.1. This class is necessary, since not every connector necessarily supports all three query types and with the connector provider it doesn't have to provide method stubs for the unimplemented methods.

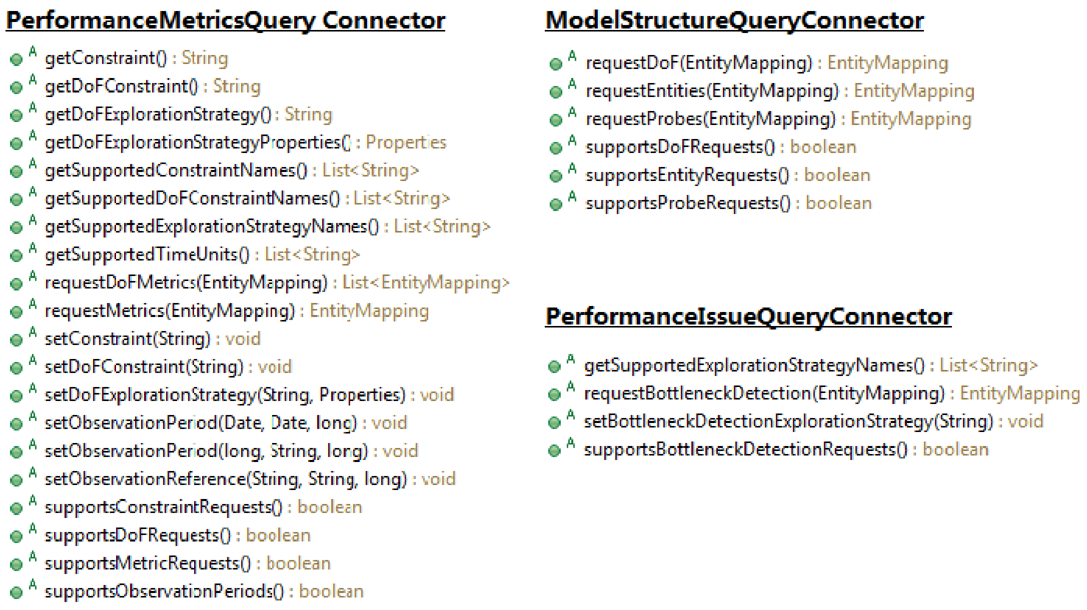


```
<<Interface>>
ConnectorProvider

+createModelStructureQueryConnectorInstance():
    ModelStructureQueryConnector
+createPerformanceIssueQueryConnectorInstance():
    PerformanceIssueQueryConnector
createPerformanceMetricsQueryConnectorInstance():
    PerformanceMetricsQueryConnector
```

Figure 2.1: The Connector Provider Interface

The core of a DQL connector are the PerformanceQueryConnector, the ModelStructure-QueryConnector and the PerformanceIssueQueryConnector. Each of these connectors handles the calculation of results for the according types of queries. In Figure 2.2 you can see the interfaces for the three connectors.



```
PerformanceMetricsQuery Connector
getConstraint() : String
getDoFConstraint() : String
getDoFExplorationStrategy() : String
getDoFExplorationStrategyProperties() : Properties
getSupportedConstraintNames() : List<String>
getSupportedDoFConstraintNames() : List<String>
getSupportedExplorationStrategyNames() : List<String>
getSupportedTimeUnits() : List<String>
requestDoFMetrics(EntityMapping) : List<EntityMapping>
requestMetrics(EntityMapping) : EntityMapping
setConstraint(String) : void
setDoFConstraint(String) : void
setDoFExplorationStrategy(String, Properties) : void
setObservationPeriod(Date, Date, long) : void
setObservationPeriod(long, String, long) : void
setObservationReference(String, String, long) : void
supportsConstraintRequests() : boolean
supportsDoFRequests() : boolean
supportsMetricRequests() : boolean
supportsObservationPeriods() : boolean

ModelStructureQueryConnector
requestDoF(EntityMapping) : EntityMapping
requestEntities(EntityMapping) : EntityMapping
requestProbes(EntityMapping) : EntityMapping
supportsDoFRequests() : boolean
supportsEntityRequests() : boolean
supportsProbeRequests() : boolean

PerformanceIssueQueryConnector
getSupportedExplorationStrategyNames() : List<String>
requestBottleneckDetection(EntityMapping) : EntityMapping
setBottleneckDetectionExplorationStrategy(String) : void
supportsBottleneckDetectionRequests() : boolean
```

Figure 2.2: The Connector Interfaces

2.2 Queueing Petri Nets

The Queueing Petri Nets (QPNs) are a modeling formalism for the qualitative and quantitative performance analysis of networks [Bau86]. For example in [Kou06b] QPNs were used to analyze the performance of a web shop. QPNs are PNs to which the queues from the QN were added. The Petri Nets (PNs) or place/transition nets are a combination of places, transitions and tokens. The places act as containers for tokens and transitions connect two or more places. They are responsible for the movement of tokens between places. In Figure 2.3 a) you can see an example for a QN. The circles represent the places `Place1`, `Place2` and `Place3`, while the black dots are the tokens contained within them. The three places are connected by one weighted transition (`Transition1`) which triggers when it can remove one token from `Place1` and three from `Place3`. Upon activation it removes these token and places two new tokens in `Place3`. Figure 2.3 b) shows the PN after the transition triggered.

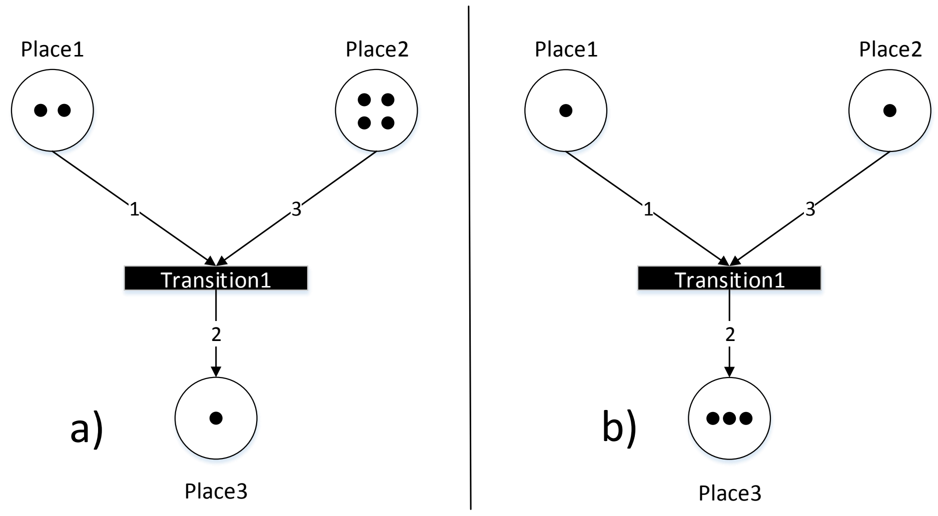


Figure 2.3: Firing of Weighted Transitions

More advanced PNs also allow for tokens of different colors and transitions that can only trigger after a certain time (timed transitions). QPNs are PNs which also contain queuing places, which are places with the functionality of a queue from the QNs. In Figure 2.4 you can see the structure of a queuing place. Incoming tokens are first placed in the waiting area, the server then moves one token from the waiting area to the depository after a specified think time. To determine which token should be moved upon activation a multitude of approaches (scheduling strategies) are available. Examples for scheduling strategies are first-come-first-served, processor-sharing, infinite server, random or priority. For more information on the subject of scheduling strategies see [SKM]. The depository of a queuing place acts as a normal QN place, meaning the tokens are available for transitions. This enables the modeler to easily represent scheduling strategies and brings the benefits of QNs into the world of Petri nets [Kou06a]. Some of the most used performance metrics for QPNs are token throughput and average token wait time for a specific place. To analyze the behaviour of tokens and to derive these values simulation can be used, as described in section 2.3

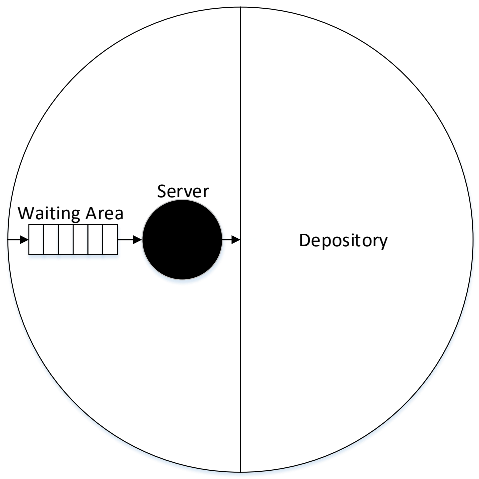


Figure 2.4: Structure of a Queuing Place

2.3 Simulation

There exist analytic approaches to analyze QPNs, but these approaches do not scale well due to the state space explosion problem [KB03a]. The solution to handling bigger networks is to simulate their behavior and analyze the statistics gathered during the simulation. Since the behavior of the model can be described as a chain of events, discrete-event simulation can be used [Mer11]. User requests and internal requests could be modeled as events. With simulations it is important to differentiate between two different types of time [Fuj99]. Simulation time is the term used to describe time inside the simulation, for example every request to database Y takes five seconds of simulation time. The simulation will not actually wait for five seconds, but maybe only a fraction of a second. This allows simulations to cover long time spans of simulated time quickly. If we want to refer to the real time passed, not the time inside the simulation we will use the term wall-clock time. When analyzing models we assume that we are interested in the long term behavior of the system. It can be assume that after a long time of simulation the behavior of the simulated system represents the long term behavior of the model. This state is called steady-state and it's behavior is called steady-state behavior. The time needed to reach the steady-state is called warm up period. There exist various techniques for the prediction of the warm up periods duration [LKPW08]. After the warm up period is over the simulation starts collecting information about the behavior of the system. The statistics to look for during steady-state analysis are the $mean\mu$, the $variance\sigma$ and the $confidenceinterval$. The confidence interval describes the interval that the mean is contained in. It can be calculated using the following formula from [Wal13]:

$$\mu(n) \pm t_{\text{n-1,1-}\alpha/2}\sqrt{\frac{S^2(n)}{n}}$$

n stands for the amount of data points, while $S^2(n)$ represents the sample variance. We can use this confidence interval to configure the run time of our simulation. We simulate not for a fixed time period, but until the confidence interval is either smaller than an absolute value or less then a prior defined percentage of the mean. These two approaches are called relative and absolute precision.

2.4 Queueing Petri net Modeling Environment

The Queueing Petri net Modeling Environment (QPME) consists of two major pieces, the QPE and simQPN, a simulation engine. QPE can be used to create QPNs via an intuitive drag-and-drop interface. QPE is entirely written in java using Eclipse's GEF. The resulting QPNs can be saved to XML or analyzed with simQPN.

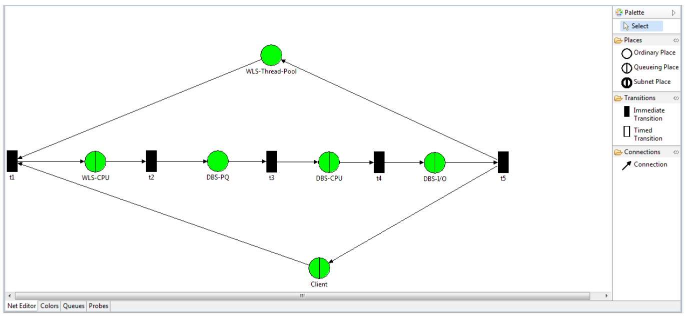


Figure 2.5: Building a QPN using QPE

SimQPN is QPME's simulator for QPN Models. It supports three approaches for the quantitative analysis of the steady-state-behavior of QPN models, the *Batch Means Method*, the *Replication/Deletion Approach* of the *Method of Independent Replications* and the *Method of Welch* [Wel67][LKK91][Paw90]. The Batch Means Method is mostly used since it is easy to configure. QPME also supports different rules for how long the simulation should run, which are *Fixed Sample Size*, *Sequential/Absolute Precision* and *Sequential/Relative Precision.* Fixed Sample Size is the default mode, which simply stops the simulation after a fixed amount of simulation time has passed. The other two approaches are sequential approaches, which means the sample size (which in this case is equivalent to the passed simulation time) isn't set prior to the start of the simulation. Instead the simulation checks during run time if a stopping rule is violated. For both modes, the stopping rule is related to the predicted precision. As the names indicate they stop the simulation once a certain absolute/relative precision is reached. The Sequential/Relative Precision rule is the easiest to use, since it doesn't require any prior knowledge. With simQPN the user can specify stat levels for each place, which indicate how much information he requires about the place. The available stat levels as listed in [SKM] are:

- **Level 0:** No statistics are collected.
- **Level 1:** Only token throughput data is collected.
- **Level 2:** Token population, token occupancy and queue utilization data is collected.
- **Level 3:** Token residence time statistics will be monitored.
- **Level 4:** A histogram of observed token residence times is added.
- **Level 5:** Token resides times are saved to a file. data

SimQPN then adjusts the precision of the tracked results accordingly to speed up the simulation.

3. Approach

This chapter explains how we plan to implement a SimQPN connector for DQL. First in section 3.1 we state the goals we want to achieve in thesis, using the approach detailed in section 3.2. We will go into detail about our approach for the implementation (subsection 3.2.1) and the evaluation of the results (subsection 3.2.2).

3.1 Goals

The implementation of the SimQPN connector for DQL can be divided in the following subgoals:

1. **Support Model Structure Queries**.This includes loading the QPN model and mapping of QPN elements to DQL services, resources and metrics.
2. **Support Performance Queries** This means we have to run a simulation and extract the requested information from the simulation results. Finally we will add support for DoF, which means running multiple simulations either parallel or sequential.
    a) **Derive SimQPN configuration from the DQL Query** To start a SimQPN-simulation a configuration is needed. This includes for example total run time, duration of the warmup period and a level indicating the required depth of statistics for each place. Choosing these values correctly is essential to collect useful information from the simulation.
    b) **Filter Simulation Results for Results** SimQPN returns the results of a simulation in form of XML-File. We plan to derive according XPaths from the requested performance metrics to extract the required information.
    c) **Add DoF support** Adding DoFsupport will allow the user to access information collected from multiple simQPN-simulations.

3.2 Approach

3.2.1 Implementation

This subsection explains the approach we took for the implementation of a SimQPN-Connector for DQL. It also explains which design choices we will have to make in the process of the implementation. Finally, we sketch our approach on the evaluation of the SimQPN Connector. As mentioned in section 2.1 the main task when implementing a DQL-Connector is to implement these three subconnectors:

1. ModelStructure
2. PerformanceMetrics
3. PerformanceIssueQuerryConnector

As mentioned in [GBK14] the best approach is to implement them in this order, since they tend to build on another. For this theses we will implement only the first two, since simQPN doesn't support anything that could be used to detect bottlenecks and building this feature from scratch is beyond the scope of this thesis.

3.2.1.1 Model Structure Queries

To answer model structure queries we need to decide what entities are available, what performance metrics can be calculated for named entities and what DoF options are available. In our case the available entities will be the places in the QPN. To extract the names and types of these places we will mostly reuse existing code from QPME. For performance metrics we decided to offer every statistic that simQPN is capable of calculating. It is important to mention that the available statistics differ between different types of places. We have two different ideas for DoF options. The first idea is to let the user change the initial population of color tokens in every place. This way the user can for example test the performance of his network for different amounts of requests. The second idea we have is to let the user define how often a place should be 'cloned'. Cloning a place will split the amounts of initial and incoming tokens equally between them. This can be used to analyze the performance of a system with for example different amounts of database servers.

3.2.1.2 Performance Metric Queries

The first decision that needs to be made when implementing support for performance metric queries is what parameters to allow for the CONSTRAINED AS clause and how to interpret these. First we want to implement only two options: 'Acurate' and 'Fast'. Simulations with 'Acurate' will run until sufficient relative precision is reached, while 'Fast' will return fast but less accurate results. The next design choice that hast to be made is how to configure the simQPN simulation. While many parameters are simple to choose sufficiently well, there are a few that make for hard decisions. The two parameters that make for the main concerns are the total runtime for simulations constrained as 'Fast' and the duration of the simulations warmup period. The total runtime for 'Fast' simulations is difficult, since the resulting precision of two QPN's with the same absolute precision can vastly differ. We plan to implement an algorithm that considers the amount of places, color tokens and number of queues in consideration. As mentioned in section 3.1 we plan to derive XPaths from every performance metric to extract the required information from the XML-files containing the simulation results. This means we have to derive the name and type of the place and the involved colors/queues from the performance metric. If there is enough time, we want to implement support for constrained as parameters of this type: 'UP TO X h Y sec' which will cause the simulation to only run for up to X hours Y seconds. Unfortunately this feature will require some changes to the existing simQPN code, since it currently only supports a total runtime measured in simulation time, not wallclock time.

3.2.1.3 Performance Issue Queries

SimQPN is a simulation engine which does not provide additonal features beyond standard simulation approaches that could be used for performance issue queries. This means that we would have to implement everything from scratch. This is beyond the scope of this thesis and will be left as future work.

3.2.2 Evaluation

Evaluating if the connector returns correct results is rather easy, we will run simulations on some QPNs from research papers via QPME and via DQL and compare the results. On the technological side of things, we plan to work with a combination of integration and unit tests. We will implement integration tests for every type of query and support these with unit tests for the key functions. A interesting idea, that unfortunately will have to be left as future work is to take the QPN, transform it to another supported modeling formalism and compare the results of the simQPN-connector to those of the corresponding connector. Since we then have two ways to produce the same results, we can use this to flexibly test the functionality of both the simQPN-connector as well as the other connector.

4. Design

When implementing a DQL connector for QPNs a lot of design decisions independent of the concrete implementation have to be made. section 4.1 explains how we mapped the QPN entities to DQL entities. Then section 4.2 explains different ways to support DoF options. In section 4.3 our implementation of constraints is explained. The decision on the duration of the warmup period is explained in section 4.4.

4.1 Mapping of QPN Entities to DQL Entities

DQL divides its entities in two categories, `resources` and `services`. This distinction exists because DQL was originally meant for architectural performance models. But since QPNs are not architectural performance models, this distinction doesn't make much sense. While we could make an arbitrary distinction into services and resources, we choose to list all entities as resources. We did this because the user would not gain anything from the distinction, it would only make things confusing for the user. We choose resources over services, since the term service might confuse some users. The next decision we had to make was which QPN entities we should use as DQL entities. When deciding which options to offer as entities, it makes sense to take a look at which kind of statistics simQPN collects, since we only want to offer entities for which there are performance metrics available. SimQPN collects statistics for the places, colors and queues. So at a first glance it would make sense to offer these three as entities, but the statistics simQPN compiles for a color are always in regards to the behavior of the color tokens within a specific place. So we moved these statistics to the places as well. While we could also add the queue statistics to the places that utilize the queue, we chose to use the queues as independent resources, since the queuing places would have too many performance metrics if we added the metrics for the underlying queues to the queuing place itself. Also there would be some weird behavior with one place using multiple queues or a queue being used by multiple queuing places.

4.2 Choice of DoF Options

For the available DoF options it would be possible to allow the user to variate almost anything. The two options that seemed the most useful were the initial population of token of every color and to allow the user to configure how often a place should be cloned. Figure 4.1 shows how the cloning would work.

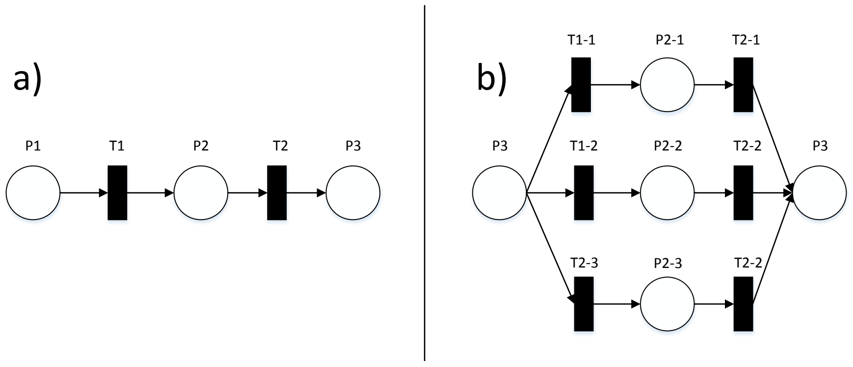


Figure 4.1: A small QPN prior and after cloning P2 two times

The cloned place would be replaced by a new set of places. For every incoming transition to the cloned place there would be a transition to every new place. All new transitions will have the same firing weight, which means it will be randomly chosen which one triggers. The outgoing transitions will be multiplied but otherwise remain the same. This way if a place is cloned X times, all incoming tokens will be equally divided between the new places, meaning every place has 1/X of the prior traffic. This feature could be used for analysis of the performance of a system with different amounts of database/application servers, by cloning the place that represents the server. We choose the initial population of token of every color over cloning because the user could use it to compare the performance of a system with different amounts of userrequests, since in QPNs userequests are usually modeled as tokens. This feature was deemed more important, since the analysis of performance with different amounts of userrequests is useful for almost any system, while the analysis of the performance with different setups doesn't always make sense. Also cloning places would require major changes to the QPN, while changing the initial population of a color token inside a place is a matter of changing one value in the QPN. Allowing the user to clone places as DoF options will be left as future work.

4.3 Implementation of DQL Constraints

Constraints in DQL are indicators for the connector how he should configure his simulation/calculations. The syntax for a constrained as clause looks as following:

```
CONSTRAINED AS 'constraint'
```

With `'constraint'` being replaced for the actual constraint. Every connector can decide which constraints it allows. The currently implemented connectors use some variation of accurate and fast as constraints, it differs between connectors how many options are available but they all rank from fastest to most accurate. We decided to only offer 'Accurate' and 'Fast' as constraints, since an implementation like 'Very Accurate' 'Accurate' 'Average' 'Fast' 'Very Fast' isn't too useful for the user, since he doesn't know what the difference between 'Very Fast' and 'Fast' is. The distinction between 'Accurate' and 'Fast on the other hand is very intuitive. This leads to the question, how do we control the accuracy/speed of the simQPN simulation? There are two ways to control the speed/accuracy of a simulation, the first one is to use a fixed sample size stopping criterion (see section 2.4. This solution has the advantage that it is very consistent. If we configure a simulation to stop after X simulation time has passed, it will always need about the same time. This means the user would know what run time he can expect for each option. The problem is, that this solution doesn't take the size or complexity of the QPN into consideration. While X seconds might be enough for reliable data about a simple small QPN, it wouldn't suffice for a large complex one. Taking the size into consideration could be done with a solution that scales the sample size according to the amount of places within the QPN. But this still leaves the issue with the complexity of the QPN not being taken into consideration. The other option is to use relative precision as stopping criterion. This approach scales with larger and more complex systems, but has the disadvantage that the time needed to reach the required precision is subject to variance. We choose this option because we prioritize reliability over predictability. A positive side effect of this implementation is that we can also allow constraints of this type:

`CONSTRAINED AS` 'Accuracy = 0.4'

This option will allow expert users freedom to configure their simulation in more detail, while the options 'Fast' (0.8 accuracy) and 'Acurate' (0.95 accuracy) are still available.

4.4 Duration of the Warmup Period

Another important parameter for the configuration of a simQPN simulation is the duration of the warmup period (see section 2.3). When this parameter is set too low, parts of the warmup period will be considered as steady state leading to incorrect results. If it is too high, the simulation takes longer than necessary. There are techniques for the prediction of the warmup periods length, from which [LKPW08] concludes MSER-5 as the favorite. We chose to not implement MSER-5 because there are plans to implement it in simQPN, so we went with a temporary solution which will be rendered obsolete when MSER-5 is implemented in simQPN. We simply set the duration of the warmup period excessively high, which leads to longer than necessary simulation. Which is preferred over sometimes choosing it too low and producing incorrect results.

5. Implementation

This chapter explains how we implemented the Model Structure Query Connector (section 5.1) and the Performance Metric Query Connector (section 5.2). In section 5.3 the exclusion of a Performance Issue Query Connector is explained

5.1 Model Structure Query Connector

Model Structure Queries return information about the structure of the underlying model. A Model Structure Query Connector offers three different types of Model Structure Queries, `LIST ENTITIES`, `LIST METRICS` and `LIST DOF`. For all of these we need to extract information about the model. QPME uses XML-files to save QPNs. In Figure 5.1 you can see an example for a small XML-file that describes a QPN. We reduced the verbosity of the qpn model to the relevant parts. This QPN contains one color (`Color1`), one queue `new Queue`), two places (`Place1`, `Place2`). In `Place1` there are initially 10 `Color1` tokens, while `Place2` starts out empty. There is one transition (`Transition`) which connects `Place1` and `Place2` and moves one token from `Place1` to `Place2` each time the transition triggers. To implement the connector we have to access information about what xml nodes exist (i.e. how many places are there?) and also information which is contained within the places attributes (i.e. initial population of a color reference). To do this we will utilize the XPath query language [CD99]. XPaths navigate through the xml tree with the parent/child relation between the object and utilizes conditions to filter the results. For example the XPath */net/places/* would return a list of nodes that contains both place elements. To filter the results xpaths uses conditions, to get all colorrefs of the Place we would use the following XPath: $/net/places/place[@id = "14235"]/color-refs/"$. The statement $place[@id = 14235]$ returns all places with the id 14235. Accessing attributes instead of nodes can be done using the following XPath structure : node/@attribute. Now that we have extracted all necessary information from the net we now need to understand how DQL saves query information. It uses the *Mapping Meta-Model*, which is described in [GBK14]. The major part of the Mapping Meta-Model is the EntityMapping class, which you can see in Figure 5.2. For our Model Structure Query Connector the `resource`, `probe` and `dof` classes have to return the requested information. The available entities will be saved as resources. Every resource has a list of probes which represent the available performance metrics for this entity. All available Degree of Freedom (DoF) options will be saved in the EntityMappings list of dofs.

```
<?xml version="1.0" encoding="UTF-8"?>
<net qpme-version="2.1.0.qualifier">
   <colors>
      <color real-color="4601ce" id="14235" name="Color1"/>
   </color>
   <queues>
      <queue name="new queue" strategy="FCFS" servers="1" id="14235>
   </queues>
   <places>
      <place id="14235" type="ordinary-place" name="Place">
         <color-refs>
            <color-ref color-id="14246" id="14237" initial-population="10"/>
         </color-refs>
      </place>
      <place id="14236" type="queueing-place" name="Queue>
         <color-refs>
            <color-ref color-id="14246" id="14238" initial-population="0"/>
         </color-refs>
      </place>
   </places>
   <transitions>
      <transition id="14239" type="immediate-transition" name="Transition">
         <connections>
            <connection id="14245" source-id="14245" target-id="14239"/>
            <connection id="14245" source-id="14239" target-id="14246"/>
         </connections>
      </transition>
   </transition>
</net>
```

Figure 5.1: Example for QPN XML file format

In 5.1 we listed the metrics we offer for ordinary places, queuing places and the queues them self. As you can see, the queues don't have any color specific metrics, only ordinary places (places that aren't queuing places) and queuing places collect these. For the queuing places there are statistics for the queuing area and for the depository (see Figure 2.4). The statistics for the depository are marked with a leading Depository_, while the statistics for the queuing area are not marked specifically. For the per Color metrics the names of the available colors are inserted. For example if a place has three colors, x1, x2 and x3 then there would be the metrics x1_ArrivalThroughput, x2_ArrivalThroughput, x3_ArrivalThroughput and so on for the metrics. The implementation of the Performance Metric Query Connector can be divided into two parts, queries without DoF (single configuration space) and queries with DoF.

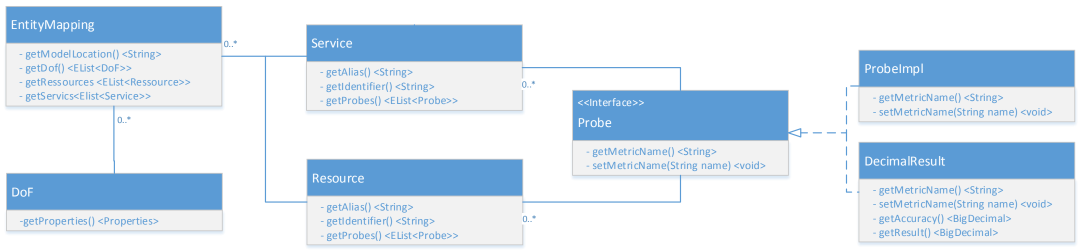

EntityMapping
- getModelLocation() <String>
- getDof() <EList<DoF>>
- getRessources <EList<Ressource>>
- getServics<Elist<Service>>
0..*
0..*
DoF
-getProperties() <Properties>
Service
- getAlias() <String>
- getIdentifier() <String>
- getProbes() <EList<Probe>>
0..*
Resource
- getAlias() <String>
- getIdentifier() <String>
- getProbes() <EList<Probe>>
0..*
<<Interface>>
Probe
- getMetricName() <String>
- setMetricName(String name) <void>
ProbeImpl
- getMetricName() <String>
- setMetricName(String name) <void>
DecimalResult
- getMetricName() <String>
- setMetricName(String name) <void>
- getAccuracy() <BigDecimal>
- getResult() <BigDecimal>

Figure 5.2: UML Diagramm of the EntityMapping

Metric	Ordinary Place	Queuing Place	Queue
Place in general			
TokenOccupancy	✓	✗	✗
Depository_TokenOccupancy	✗	✓	✗
MeanTokenResidenceTime	✗	✗	✓
QueueUtilization	✗	✗	✓
QueueUtilizationDueToThisPlace	✗	✓	✗
TotalArrivalThroughput	✗	✗	✓
TotalDepartureThroughput	✗	✗	✓
per Color			
ArrivalThroughput	✓	✓	✗
DepartureThroughput	✓	✓	✗
MaximumTokenPopulation	✓	✓	✗
MaximumTokenResidenceTime	✓	✓	✗
MeanTokenPopulation	✓	✓	✗
MinimumTokenPopulation	✓	✓	✗
StandardDeviationTokenResidenceTime	✓	✓	✗
TokenColorOccupancy	✓	✓	✗
Depository_ArrivalThroughput	✗	✓	✗
Depository_DepartureThroughput	✗	✓	✗
Depository_MaximumTokenPopulation	✗	✓	✗
Depository_MaximumTokenResidenceTime	✗	✓	✗
Depository_MeanTokenPopulation	✗	✓	✗
Depository_MeanTokenResidenceTime	✗	✓	✗
Depository_MinimumTokenPopulation	✗	✓	✗
Depository_MinimumTokenResidenceTime	✗	✓	✗
TokenColorOccupancy	✗	✓	✗

Table 5.1: Available Performance Metrics for the different types of places

5.2 Performance Metric Query Connector

5.2.1 Single Configuration Space

Single configuration space queries are queries without DoF. Running single configuration space queries can be divided into three tasks, first starting the simulation, then extracting information from the results and finally returning the information. To start a simQPN simulation there is a clear defined interface that has to be called. It requires that the simQPN configuration is already saved to the QPN file. In chapter 4 we already discussed how the precision, maximum run length and duration of the warmup period will be chosen. Aside from these values, there are the stat levels for the places that have to be defined. SimQPN allows us to assign stat levels to each place, which indicate what statistics are collected for each place. There are six different stat levels available [SKM]:

- **Level 0:** No statistics are collected
- **Level 1:** Only token throughput data, aka arrival/departure throughput
- **Level 2:** Token occupancy, queue utilization, token population statistics (maximum token population, mean token population, ...)
- **Level 3:** Token residence time statistics (maximum token residence time, mean token residence time, ...)
- **Level 4:** Histogram of token residence times
- **Level 5:** Token residence times are saved to an additional file

For our connector the levels four and five are useless, since we have no way of returning the collected data. From the other four levels we always choose the lowest one, that still collects sufficient information. This speeds up the simulation, since as little information as possible is collected. The stat levels for the queue statistics are applied to the queuing place that utilizes the queue. SimQPN saves the results of a simulation to a xml file, this means we can use XPath again to retrieve the required information. From the name of the resource and the performance metric enough information can be collected to formulate exact XPaths to extract the answers to the query. To return the information to DQL the ProbeImpl for every performance metric is replaced with a DecimalResult (see Figure 5.2). We set the accuracy of the Decimal result to the value we used for the configuration. We parse the simQPN log file for errors, and if an error occurred during the simulation the accuracy is set to zero to inform the user that the collected results might be incorrect.

5.2.2 Degrees of Freedom

The analysis of queries with DoF comes down to running multiple simulations, so the first question is if they should be run parallel or sequential. simQPN does offer support for the parallel execution of simulations. But even though this method is faster we decided against it, since there are some QPN features where parallel execution is not supported yet. Again we chose to cover all QPNs rather than focusing on execution speed. For every simulation of a DoF query minor changes have to be made to the QPN. We read the original QPN, make the changes and then write it to a folder containing log files, qpn files and result files for the simulation. This allows the user to track what changes were made and what the results were. Returning the information about the results is convenient, since DQL expects a list containing the results of every simulation. The information about which DoF parameters were chosen for which simulation is saved as DecimalResults (see Figure 5.2). The name of the DoF parameter (i.e. initial population of x1 tokens in Client) is saved as the DecimalResults name, the value of the parameter as the value of the DecimalResult. Accuracy is set to 1.0 since DQL does not allow empty fields.

5.3 Performance Issue Query Connector

As mentioned in subsubsection 3.2.1.3 we will not implement support for a performance issue queries, so there is no need to implement a Performance Issue Query Connector.

6. Evaluation

In this chapter, we evaluate the effect of constraints on the performance of a query in section 6.1. The validation of our implementation is described in section 6.2.

6.1 Evaluation of constraints

Since we choose relative precision as a stopping criterion, the total run time of the simulations is subject to variance. This makes the evaluation of the `CONSTRAINED AS` feature difficult, since we can not simply run two simulations with different accuracy and check if the simulation with lower accuracy. To combat this we ran extensive performance test. We choose two models, a small QPN and a large QPN and ran 10 simulations on each, with an accuracy of 0.95 and 0.7. This lead to a total of 40 simulations. As models we choose the ispass03.qpe and SjAS04Model_6AS-L5.qpe which are shipped with QPME (see [KB03b][**?**]). For the performance tests we used a server with the following specifications:

- **Processor:** Intel Core i5 4690K 4x 3.50GHz So.1150
- **RAM:** 8GB G.Skill TridentX DDR3-2400 DIMM CL10
- **Videocard:** 4096MB MSI GeForce GTX 970 OC Aktiv PCIe 3.0 x16
- **Motherboard:** ASRock Z87 Pro3 Intel Z87 So.1150 Dual Channel DDR3 ATX

We choose to only test the performance of single configuration space queries, since queries with DoF are a composite of queries with single configuration space, so if the constraints work for queries without DoF then it will work for queries with DoF as well. In 6.1 you can see the results of our tests.

To determine that the test results follow a normal distribution we used the lilliefors test. More information about this test can be found in [AM07]. MATLAB offers the convenient function `lillietest(x)` for the execution of lilliefors tests, which returns 1 if the test rejects the null hypothesis that the data comes from a distribution in the normal family at the 5% significance level. As you can see in Table 6.1 all four series of measurements are normal distributed according to the liliefors test. Now that we know that the samples follow a normal distribution we can use the student's t test [Hsu38] to determine whether the runtime of the ten runs for each model with an accuracy of 0.98 can be from a normal distribution with the same mean as the runs with an accuracy of 0.7. Again MATLAB offers the convenient function `testt2(x, y)` to run a two-sample t test. It returns 1 if

Model	ispass03	ispass03	SjAS04Model_6AS-L5	SjAS04Model_6AS-L5
Accuracy	0.98	0.7	0.98	0.7
Duration 1st Run [s]	1.606	1.546	26.702	22.23
Duration 2nd Run [s]	1.661	1.551	26.598	22.246
Duration 3rd Run [s]	1.65	1.532	26.395	22.167
Duration 4th Run [s]	1.671	1.502	26.659	22.183
Duration 5th Run [s]	1.652	1.574	26.493	22.23
Duration 6th Run [s]	1.648	1.592	26.534	22.138
Duration 7th Run [s]	1.642	1.535	26.522	22.245
Duration 8th Run [s]	1.643	1.539	26.382	22.183
Duration 9th Run [s]	1.635	1.536	26.577	22.106
Duration 10th Run [s]	1.668	1.543	26.565	22.184

Table 6.1: Results of the performance tests

the hypothesis that both series of data follow the same normal distribution is rejected at a 5% significance level and 0 otherwise. The student's t test returns 1 for the small and the large model as shown in Table 6.1, meaning the runtime of the runs with different accuracy can not follow the same normal distribution. This proves, that the constrained as feature correctly sets the accuracy.

Model	ispass03	ispass03	SjAS04Model_6AS-L5	SjAS04Model_6AS-L5
Accuracy	0.98	0.7	0.98	0.7
Liliefors test result	0	0	0	0
Liliefors test p-value	0.44	0.29	>0.5	0.33
Student's t test result	1		1	
Student's t test p-value	3.6 E-09		9.80E-28	

Table 6.2: Results of the performance tests

6.2 Validation

The validation of our simQPN connector is based on two assumptions:

- The DQL QEE works correctly
- SimQPN works correctly

The consequences of these assumptions are, that we do not need to test if results returned by the QEE/simQPN are correct. This means we do not need to test if for example aggregates work or if the results returned by simQPN are correct. What we do need to test is:

1. Are the correct DQL enties listed?
2. Are the correct metrics listed?
3. Are the correct DoF options listed?
4. Are the stat levels set correctly?
5. Is the simQPN configuration correct?
6. Are the simQPN results extracted correctly?
7. Are the QPNs modified correctly for queries with DoF?

With these questions in mind we designed a set of integration and unit tests. Questions 1-3 are covered by integration tests for the model structure query connector. A parameterized test verifies that `LIST ENTITES`, `LIST METRIC` and `LIST DOF` queries for a set of QPNs return the correct results. Unit tests that use XPath to retrieve information about the stat levels and simQPN configuration from the modified QPN verify that the points four and five work correctly. Another unit test checks whether the correct results are extracted from a simQPN results file. Question seven is also covered by a unit test, that checks the properties of the QPN files after they were modified for a DQL query. The test suite is completed by integration tests for different performance metric queries on the same set of QPNs we used for the model structure queries. This set of test covers the above mentioned error sources and offers good general coverage as well to ensure that the connector is working as intended.

7. Conclusion

In this final chapter, we provide a summary and conclusion of this thesis in section 7.1 and point out options for future work in section 7.2. These options include optimization for the query processing, ideas for case studies and additional features.

7.1 Summary and Conclusion

This thesis develops a DQL connector for QPN analysis using the SimQPN simulation engine. This connector allows users to analyze QPNs with simQPN simulation through the DQL interface. This means users can now start simulations with next to no prior knowledge about simulations, since our connector takes care of the simulations configuration. A simulation usually returns all information that was collected and the user has to find the values he is looking for. This is different with DQL since DQL filters the results to fit the query. Aside from improving the usability of simQPN, the simQPN connector for DQL also adds new features to simQPN, due to the support for the exploration of DoF options. To implement a simQPN connector for DQL we had to implement a model structure query connector and a performance metric query connector. While the model structure query connector simply has to extract information from the QPN and return it, the performance metric query connector is far more complex. It has to start and configure simulations, filter the simulations results and support the exploration of DQL options. Before we could implement the subconnectors, design decisions had to be made. We had to decide how many and which DQL entities we offer and whether to brand them as resources or services. We choose to offer all places and queues of the QPN as resources for lucidity reasons. Next we had two options for what DoF options we would offer, cloning of a place (see section 4.2) or initial color token population. We went with the initial color token population over the cloning since the variation of initial color token population can be used for a broader spectrum of purposes. DQLs performance metric queries contain a `CONSTRAINED AS 'value'` clause, who's value will be directly transmitted to the connector. This feature should be used to allow the user to regulate the speed/accuracy of the simulation. We choose to allow three options for constraints, 'Accurate', 'Fast' and 'Accuracy = X' with $0 < X < 1$. 'Accurate' and 'Fast' are equivalent to 'Accuracy = 0.95' and 'Accuracy = 0.8' respectively. We used a relative precision stopping criterion to determine the length of the simulation, which means to implement these constraints we had to adjusted the value of the stopping criterion accordingly. On the implementation side we used DQLs EntityMapping class to return the answer to queries. The component based nature of the EntityMapping makes it possible to return the answers to different queries through the same interface. To answer model structure queries we retrieve information about the QPN from the xml file using XPath, to answer performance metric queries we run simQPN simulations using the according interface. To run simQPN simulation we first had to write the simQPN configuration to the QPN file. The simQPN configuration was derived from the constrained as clause (accuracy) and the stat levels were set to retrieve just enough information to answer the queries. Since simQPN saves the results of a simulation as xml files we could utilize XPaths to retrieve the required information. To evaluate if the simQPN connector works correctly we used a set of integration and unit tests which we developed in regard to the possible error sources. Since we could not utilize automated testing to evaluate whether the constrained as feature works correctly due to the run times variance we ran a performance test to show the impact of the constrained as feature on the run time. This evaluation allows us to be certain that we achieved our goals and objectives. The simQPN connector improves simQPNs usablilty for new users and added a new feature with the support for DoF. We successfully showed that DQL can be used for modeling formalisms that are not designed for design time analysis.

7.2 Future Work

This section list the ideas that did not make it into this thesis but could be implemented to improve the simQPN connector.

- Additional Features
  - **Implementation of cloning places as DoF parameter:** As explained in section 4.2 letting the user clone places within the QPN has many uses. This feature should be prioritized since it yields the greatest benefits.
  - **Integrate an DQL user interface in QPME:** A specialized interface that doesn't offer full DQL support, but only the relevant parts for running DQL queries on QPNs would lower the barrier of entry for new users and show the advantages of DQL even when working with only one modeling formalism.
  - **Support for Performance Issue Queries:** The automated detection of bottlenecks would be a great step toward self-aware computing, but still requires a lot of work. It is up for disscussion wether this feature should be implemented in simQPN, the connector or DQL.
- Optimizations for query processing
  - **Use parallel simulation for DoF queries:** This feature would speed up the processing of DoF queries, but unfortunately it can not be implemented until simQPN supports parallel simulation of every QPN.
  - **Implementation of MSER-5 to detect the duration of the warmup period:** This feature would speed up the simQPN connector, since it currently overestimates the duration of the warmup period. This feature should not be implemented in the connector but in simQPN directly since the users of QPME could also profit from it.
- Case Studies
  - **Use model transformation to cross test two connectors:** As explained in subsection 3.2.2 automated model transformation between QPN and another modeling formalism that DQL supports could be used to compare the results of two simulations on the same system. This would allow automated integration testing for both connectors. This feature is on hold until a fully automated model transformation is available.

8. Acronyms

PN Petri Net

QN Queueing Net

QPN Queueing Petri Net

PCM Palladio Component Model

DQL Descartes Query Language

DML Descartes Modeling Language

QPME Queueing Petri net Modeling Environment

QPE Queueing Petri net Editor

OSGi Open Services Gateway Initiative

DoF Degree of Freedom

QEE Query Execution Engine

SQL Structured Query Language

List of Figures

List of Tables

Bibliography

[AM07] H. Abdi and P. Molin, “Lilliefors/van soest’s test of normality,” *Encyclopedia of measurement and statistics*, pp. 540–544, 2007.

[AME$^+$09] A. Agarwal, J. Miller, J. Eastep, D. Wentziaff, and H. Kasture, “Self-aware computing,” *DTIC Document*, 2009.

[Bau86] F. Bause, “Funktional und quantitativ analysierbare rechensystemmodelle; über verbindungen von petrinetz- und warteschlangen-modellwelten,” 1986.

[Bau93] ——, “Queueing petri nets-a formalism for the combined qualitative and quantitative analysis of systems,” pp. 14–23, 1993.

[BDMIS04] S. Balsamo, A. Di Marco, P. Inverardi, and M. Simeoni, “Model-based performance prediction in software development: A survey,” *Software Engineering, IEEE Transactions on*, vol. 30, no. 5, pp. 295–310, 2004.

[BHK11] F. Brosig, N. Huber, and S. Kounev, “Automated extraction of architecture-level performance models of distributed component-based systems,” pp. 183–192, 2011.

[CD99] J. Clark and S. DeRose, “Xml path language (xpath) version 1.0,” 1999.

[Fuj99] R. M. Fujimoto, “Parallel and distribution simulation systems,” p. 320, 1999.

[GBK14] F. Gorsler, F. Brosig, and S. Kounev, “Performance queries for architecture-level performance models,” pp. 99–110, 2014.

[Hsu38] P. Hsu, “Contribution to the theory of” student’s” t-test as applied to the problem of two samples,” *Statistical Research Memoirs*, 1938.

[KB03a] S. Kounev and A. Buchmann, “Performance modelling of distributed e-business applications using queuing petri nets,” pp. 143–155, 6-8 March 2003 2003.

[KB03b] ——, “Performance modeling of distributed e-business applications using queueing petri nets,” in *Proceedings of the 2003 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS 2003), Austin, Texas, USA, March 6-8, 2003.* Washington, DC, USA: IEEE Computer Society, 2003, pp. 143–155, best-Paper-Award.

[KB06] ——, “Simqpn—a tool and methodology for analyzing queueing petri net models by means of simulation,” *Performance Evaluation*, vol. 63, no. 4, pp. 364–394, 2006.

[Kou06a] S. Kounev, “Performance engineering of distributed component-based systems,” 2006.

[Kou06b] ——, “Performance modeling and evaluation of distributed component-based systems using queueing petri nets,” *IEEE Computer Society*, 2006.

[Kou06c] ——, “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, July 2006.

[LKK91] A. M. Law, W. D. Kelton, and W. D. Kelton, “Simulation modeling and analysis,” 1991.

[LKPW08] K. Ley and J. K. Preston White, “Simulation output analysis: Automation of truncation in arena,” *School of Engineering and Applied Science University of Virginia, Charlottesville*, 2008.

[Mer11] P. Merkle, “Comparing process- and event-oriented software performance simulation,” *Karlsruhe Institute of Technology (KIT)*, 2011.

[Paw90] K. Pawlikowski, “Steady-state simulation of queueing processes: survey of problems and solutions,” *ACM Computing Surveys (CSUR)*, vol. 22, no. 2, pp. 123–170, 1990.

[PR10] C. J. Puigjaner and Ramon, “From the origins of performance evaluation to new green ict performance engineering,” 2010.

[SKM] S. S. Samuel Kounev and P. Meier, “Qpme 2.0 - a tool for stochastic modeling and analysis using queueing petri nets,” *Karlsruhe Institute of Technology.*

[Wal13] J. C. Walter, “Parallel simulation of queueing petri net models,” 2013.

[Wel67] P. Welch, “The use of fast fourier transform for the estimation of power spectra: a method based on time averaging over short, modified periodograms,” *IEEE Transactions on audio and electroacoustics*, pp. 70–73, 1967.