Karlsruhe Institute of Technology

Automated Transformation of Palladio Component Models to Queueing Petri Nets

Diploma Thesis of

Philipp Meier

At the faculty of Computer Science
Institute for Program Structures
and Data Organization (IPD)

Reviewer:	Prof. Dr. Ralf H. Reussner
Advisor:	Dr.-Ing. Samuel Kounev
Second advisor:	Dr.-Ing. Heiko Koziolek

April 20, 2010 – October 20, 2010

Abstract

In today's software engineering landscape, performance and scalability properties of systems are of crucial importance to ensure that quality-of-service requirements are satisfied. Changing systems in late development stages is very costly and therefore performance predictions early in the development process are essential to detect potential problems before resources have been spent on implementation. The Palladio Component Model (PCM) is a domain-specific modeling language for component-based systems enabling performance prediction at design time. Four performance-influencing factors are modeled for each system component: the component implementations, the external services they use, the execution environment on which they are deployed, and the component usage profiles. The modeled system is analyzed for selected performance metrics such as response time, throughput and resource utilization, by means of a PCM model solver. Several solvers exist which place different restrictions on the PCM model instance and offer different trade-offs between accuracy and overhead. However, existing solvers offer limited flexibility in terms of efficiency and accuracy of the solution process, and suffer from scalability issues. Queueing Petri Nets (QPNs) are another general-purpose modeling formalism, at a lower level of abstraction, that has been shown to lend itself very well for performance analysis of distributed component-based systems. Efficient and mature solution techniques are available for QPN models and therefore an automatic transformation from PCM to QPN models is highly desirable. It would open up the benefits of QPNs to the PCM community and provide a basis for future transformation to QPNs from other source models in the performance engineering domain. This thesis provides a bridge between the PCM and QPN formalisms making the following specific contributions: i) A formal mapping from PCM to QPNs analyzing the feasibility of using QPN models as a target analysis formalism for PCM models, ii) Implementation of an automatic transformation from PCM to QPNs in the form of a new PCM solver tool based on SimQPN, a mature simulator for QPNs, iii) An extensive evaluation of the PCM-to-QPN transformation in terms of results accuracy and analysis overhead, iv) A detailed comparison of the new SimQPN solver with existing PCM solvers, v) Formulation of future research directions, especially regarding the PCM stochastic expressions language and the possibilities to reduce expressions to more commonly-known probability distributions. The new SimQPN solver proved to be much faster than the existing SimuCom reference solver with performance improvements of up to 20 times. In most cases, the provided results were very accurate with a deviation from the reference values below 15%. The tool was integrated into the PCM-Bench tool, delivered with the PCM meta-model and compared with the SimuCom reference solver as well as with LQNS and LQSim, two existing solvers based on layered queueing networks. Customized PCM instances were created for each of the mapped features, evaluating for the first time in detail, the PCM features supported by each solver. Additionally, to evaluate the transformation in realistic conditions, five case studies were conducted using the largest existing PCM instances that could be obtained. One of the case studies was conducted in cooperation with ABB Research, demonstrating the applicability of the results of the thesis in an industrial context.

Zusammenfassung

In der heutigen Softwareentwicklung sind Performanz und Skalierbarkeit eines Systems wichtige extrafunktionale Eigenschaften. Weil späte Änderungen im Entwicklungsprozess sehr konstenintensiv sind, ist eine Performanzvorhersage früh im Entwicklungsprozess notwendig. Sie ermöglicht es, potentielle Probleme zu erkennen, bevor Ressourcen in die Implementierung investiert wurden. Das Palladio-Component-Model (PCM) ist eine domänenspezifische Modellierungssprache für komponentenbasierte Systeme zur Performanzvorhersage zur Entwurfszeit. Ein mit PCM modelliertes System kann für ausgewählte Performanzmetriken wie Antwortzeit, Durchsatz, und die Auslastung von Ressourcen mittels eines PCM-Modell-Solvers analysiert werden. Verschiedene Solver existieren, die unterschiedliche Einschränkungen bezüglich der zu analysierenden Modelle aufweisen, und die unterschiedliche Trade-Offs zwischen Genauigkeit und Analyseaufwand ermöglichen. Verfügbare Solver bieten jedoch beschränkte Flexibilität, was die Effizienz und Genauigkeit des Löseverfahrens angeht, und leiden unter Skalierbarkeitsproblemen. Queueing Petri-Netze (QPNs) sind ein weiterer, universeller Modellierungsformalismus auf einer niedrigeren Abstraktionsebene als PCM. Sie haben sich für die Performanzanalyse von verteilten Software-Systemen bewährt und bieten effiziente und ausgereifte Lösungsverfahren verfügbar. Eine automatisierte Transformation von PCM zu QPNs ist daher das Ziel dieser Arbeit. Die Vorteile von QPNs würden so für die PCM-Gemeinschaft zugänglich gemacht und die Transformation würde eine Basis für weitere Transformationen zu QPNs bilden. Diese Diplomarbeit bildet eine Brücke zwischen den PCM und QPN Formalismen und leistet die folgenden Beiträge: i) Eine formale Abbildung von PCM zu QPNs, die die Möglichkeit der Benutzung von QPNs als Analyseformalismus für PCM analysiert, ii) Implementierung einer automatisierten Transformation von PCM zu QPNs in Form eines neuen Solvers basierend auf SimQPN, einem ausgereiften Simulator für QPNs, iii) eine ausführliche Auswertung der PCM-zu-QPN-Transformation bezüglich der Genauigkeit und dem Analyseaufwand, iv) ein detaillierter Vergleich des neuen SimQPN-Solvers mit existierenden PCM-Solvern, v) Formulierung neuer Forschungsziele, insbesondere bezüglich der PCM-Stochastic-Expressions-Sprache, und den Möglichkeiten, Expressions in verbreitete Wahrscheinlichkeitsverteilungen zu reduzieren. Der neue SimQPN-Solver erzielte um bis zu zwanzigfach reduzierte Analyselaufzeiten im Vergleich zur bestehenden Referenz, dem SimuCom-Solver. Die erzielten Vorhersagen waren in den meisten Fällen sehr genau mit einer Abweichung von unter 15% von den Referenzwerten. Der Solver wurde in das PCM-Bench-Werkzeug integriert, das mit dem PCM-Metamodell ausgeliefert wird. Neben SimuCom wurde der Solver auch mit LQNS und LQSim verglichen, zweier auf Layered-Queueing-Networks basierenden Solvern. Für jedes abgebildete Feature wurden individuell angepasste PCM-Instanzen erstellt, die erstmals eine detaillierte Auswertung der von den Solvern unterstützten Features ermöglichten. Zusätzlich wurden fünf Fallstudien mit den größten bestehenden PCM-Instanzen, die für die Diplomarbeit verfügbar waren, durchgeführt, um die Transformation unter realistischen Bedingungen auszuwerten. Eine der Fallstudien wurde in Kooperation mit ABB Research durchgeführt, was die Anwendbarkeit der Ergebnisse der Diplomarbeit in einem industriellen Umfeld demonstriert.

Contents

List of Figures

List of Tables

1. Introduction

In today's software engineering landscape the extra-functional properties (e.g., performance, availability and reliability) of systems are of great importance. A lot of research has been dedicated to developing methods and strategies to evaluate these properties at system design time in order to ensure that systems meet their quality-of-service requirements (e.g., ATAM [KKC00]). One of the most important extra-functional properties is performance, which will be the focus of this thesis. The software architecture has a great effect on the system performance and is available early in the development cycle [RH08]. Architectural changes in the late development stages are very costly and therefore it is essential to be able to predict the system performance at system design time in order to detect potential problems before resources have been spent on implementation. The Palladio Component Model (PCM) [BKR09] is a domain-specific modeling language for component-based systems enabling performance predictions at design time. Four performance-influencing factors are modeled for each system component: the component implementations, the external services they use, the execution environment on which they are deployed, and the component usage profiles. The modeled system is analyzed for selected performance metrics such as response time, throughput and resource utilization, by means of a PCM model solver. Several solvers exist which place different restrictions on the PCM model instance and offer different trade-offs between accuracy and overhead. The advantage of PCM is that it is formally defined and provides flexibility in both creating and analyzing model instances.

1.1 Motivation

PCM models are analyzed through a transformation to a predictive performance model at a lower level of abstraction. In the case of SimuCom, the reference solver distributed with the PCM meta-model, the transformation targets Java sourcecode based on a general-purpose simulation framework. Other existing transformations are a transformation to Layered Queueing Networks (LQNs) [KR08] and a transformation to Stochastic Regular Expressions [Koz08, pp. 199 ff.]. Based on these transformations, several PCM solvers have been developed which place different restrictions on the PCM model instance and offer different trade-offs between accuracy and overhead. However, existing solvers provide limited flexibility in terms of efficiency and accuracy of the solution process, and suffer from scalability issues. Recent efforts in using performance models for performance management at run-time place new requirements on the flexibility and efficiency of the solving process

[KBHR10]. This diploma thesis explores a new analysis method for PCM that aims at obtaining comparable results accuracy to that of existing methods, in less time.

Queueing Petri Nets (QPNs) are another general-purpose modeling formalism, at a lower level of abstraction, that has been shown to lend itself very well to modeling and analyzing the performance of distributed component-based systems [Kou06, KB03]. Both a mature and optimized simulation engine (SimQPN [KB06] which is part of QPME – the Queueing Petri net Modeling Environment [KD09]), as well as analytical techniques (e.g., HiQPN-Tool [BBK95]), are available for solving QPN models. However, QPNs are a general-purpose modeling formalism and therefore have no constructs for representing software domain elements like components or system usage profiles directly. They are defined at another level of abstraction and a mapping from the components of the system to the appropriate QPN model has to be developed manually and individually for each project. It would therefore be desirable to be able to model the system in PCM, which supports most system entities directly, but conduct the analysis using the available tools and methods for QPNs, in particular the highly optimized SimQPN simulator [KB06]. An automatic transformation from PCM to QPN models would make this possible opening up the benefits of QPNs to the PCM community and providing a basis for future transformation to QPNs from other source models in the performance engineering domain. The benefits of such a transformation include:

Ease of Modeling Software systems can be modeled using PCM-Bench [BKR09], which has been specifically designed for modeling the performance-relevant aspects of software systems. PCM models are much more accessible to software engineers and easier to maintain than QPN models.

Faster Analysis The SimQPN [KD09] simulator is optimized for QPN analysis and is expected to run faster than SimuCom [BKR09], which is built on top of a generic simulation framework.

Better Scalability With higher performance comes the ability to analyze larger PCM models, improving scalability. Furthermore, a parallelization of SimQPN is planned [KD09], which would further improve the performance and scalability in the future.

Tool Extensibility The PCM to QPN transformation can be reused with minor modifications as further analysis methods for QPNs become available.

Customizable Simulation SimQPN offers the ability to configure what data exactly to collect during the simulation and what statistics to provide at the end of the run [KD09]. This allows to customize and fine-tune the simulation to the specific requirements of the analysis which can significantly reduce the simulation overhead.

Extended Results Due to the different target formalism some result metrics might be easier to obtain compared to other analysis methods and can be offered without much implementation effort.

Future Transformations The transformation implementation and documentation can be used as a starting point for other transformations that target QPNs.

1.2 Aim of the Thesis

The primary goal of the thesis is to evaluate the usefulness of QPNs as a target analysis formalism for PCM. This includes the derivation of a formal mapping from PCM to QPNs and the implementation of a prototypical solver tool used to evaluate the mapping. The solver tool is implemented using a model-to-model transformation from PCM to QPNs. The QPNs are analyzed using the SimQPN simulator [KB06].

The central questions to be answered are:

- Does the expressiveness of QPNs allow the complete PCM meta-model to be mapped? Which limitations apply?
- Is it possible to obtain simulation results comparable to those of the existing SimuCom simulator and LQN solver provided with the PCM-Bench?
- Is it possible to obtain the simulation results faster? For which scenarios does this apply?
- What are the trade-offs provided by the existing solvers and the new SimQPN solver?

1.3 Outline

Chapter 2 starts with an introduction to the foundations of this thesis. The approach followed to reach the goals of the thesis is presented in Chapter 3. Related work is covered in Chapter 4. Chapter 5 presents the formal PCM-to-QPN mapping and its limitations. The solver tool implementing the mapping transformation is presented in Chapter 6. The evaluation setup and feature by feature evaluation results are presented in Chapter 7. The case studies are presented in Chapter 8. Finally, the conclusions, contributions, and future work are discussed in Chapter 9.

2. Foundations

In this section, the main concepts and technologies that this diploma thesis builds on are introduced. In Section 2.1, we start with an introduction to the Palladio Component Model (PCM), which is both the host technology and source meta-model for the transformation to be developed. Queueing Petri Nets, the target model for the transformation, and related variants, are introduced in Section 2.2. The target tool for analyzing Queueing Petri Nets, the Queuing Petri net Modeling Environment (QPME), is introduced in Section 2.2.5. In Section 2.3 model-to-model transformations are introduced and major related technologies are listed. Section 2.3.2 provides an introduction to QVT Operational.

2.1 Palladio Component Model (PCM)

2.1.1 Overview

The Palladio Component Model (PCM) is a meta-model allowing the specification of performance-relevant information of a component-based architecture [BKR09]. It focuses on the software performance engineering (SPE) and component based software engineering (CBSE) domains. Four factors essentially determine the performance of a software component [BKR09]: its implementation, the performance of external services it requires, the performance of the execution environment it is deployed on, and the usage profile.

In a large software project there is usually not a single person that has sufficient information about all these factors to capture them in a PCM model instance. Furthermore, the information is either not available at all times or it is not fixed. For this reason, PCM allows the performance-relevant information of a component to be specified using parametric dependencies. This flexibility is used to provide a custom domain specific language for each of the four roles in the Component-Based Software Engineering (CBSE) development process [BKR09]: component developer, software architect, system deployer and business domain expert. Each role contributes their part of the information, which are then assembled to a complete model at the time of analysis of the PCM instance. Each of the four parts are briefly introduced in the following sections.

The final section briefly introduces PCM stochastic regular expressions.

The PCM meta-model is formally defined and well-specified. The ecore meta-model of the Eclipse Modeling Framework (EMF) [emf] is used. This allows for flexibility in both creating a PCM model instance and in using that instance for analysis. An editor called the

PCM-Bench implements the mentioned custom domain specific languages for the CBSE development roles. However, creating the PCM instance by hand is not the only option. For example, a recently developed method allows the automatic extraction of a PCM instance from a running system [BKK09]. On the analysis side a number of methods are already available. The main analysis method that currently supports the largest number of PCM modeling constructs is SimuCom [BKR09]. It uses generated Java code to simulate the PCM instance. Other options for analysis include a transformation to Layered Queueing Networks [KR08] as well as to Stochastic Regular Expressions [Koz08, pp. 199].

2.1.2 Repository

The component developer specifies the implementation-specific information of a component and stores it into a component repository. After specifying the provided and required interfaces of a component, a service effect specification (SEFF) is specified for each of the provided interface signatures. The SEFFs abstractly model the externally visible behavior of a service with resource demands and calls to required services [BKR09].

2.1.3 System

The system architect uses the component specifications of the component developer to assemble the system. Like a component, the system has provided and required interfaces, which represent the boundaries of the modeled system. In between, components are assembled by referencing their specification in the component repository. References to components with a matching providing and requiring interface can be connected. The component references are called assembly contexts. This way the software architect can choose which components to use without knowing any implementation details.

2.1.4 Resource Environment and Allocation

The system deployer uses his knowledge about the target runtime system to model the resource environment. The resource environment is divided into resource containers which each can have a number of different resource types. For each assembly context, representing a runtime instance of the component, the system deployer specifies the resource container that instance is deployed on. This deployment part of the model is called allocation. Consequently, any resource demands specified in the SEFF of the referenced component logically occupy the resources of the resource container the assembly context is deployed on.

2.1.5 Usage Model

The domain expert specifies the final piece of information: the usage profile. For each of the system provided interfaces it is specified, how often, and with which input parameters, the service is called. For this, stochastic probability distributions can be used to model real life scenarios.

2.1.6 Stochastic Expressions (StoEx)

The PCM stochastic expressions language is used to specify arbitrary discrete and continuous stochastic expressions. Every PCM *RandomVariable* contains a stochastic expression specification, which is a valid instance of the StoEx language. At runtime, the specification is parsed and an abstract root entity *Expression* is returned. There are concrete sub-entities of *Expression* for numbers and other literals, common probability functions like a probability mass function (PMF) and exponential function, and entities for combining or modifying other *Expression*s. A *Product*-expression, for example, references a *left* and a *right Expression* and applies a product operator (like multiplication).

A detailed examination and a description of the underlying meta-model can be found in [Koz08, pp. 96].

2.2 Petri Nets

Petri nets are a family of formalisms that originated from what are now called ordinary Petri Nets. Over time, the basic concept was extended to allow more complex situation to be modeled. The following sections give a brief overview of the different evolution stages leading up to Queueing Petri Nets (QPNs). Finally, QPME, a tool for modeling and analysis of QPNs, is introduced.

2.2.1 Ordinary Petri Nets

An ordinary Petri Net is a bipartite, directed graph. It consists of one set of places and one set of transitions. Places are connected to transitions, and transitions to places, but not among themselves. Places contain a certain number of tokens. The number of tokens at the start of the analysis is determined by an initial marking function. The forward incidence function defines how many tokens a transition requires in each connected place to be ready to fire. When a transition fires, it deducts that number of tokens from each incoming place and deposits new tokens in other places if any backward incidence functions are defined. If more than one transition is ready to fire, one is randomly chosen with equal probability. A formal definition [Bau93] is given below:

Definition 1 *An ordinary Petri Net (PN) is a 5-tuple $PN = (P, T, I^-, I^+, M_0)$, where:*

1. $P = \{p_1, p_2, \ldots, p_n\}$ *is a finite and non-empty set of* places*,*
2. $T = \{t_1, t_2, \ldots, t_m\}$ *is a finite and non-empty set of* transitions*,* $P \cap T = \emptyset$*,*
3. $I^-, I^+ : P \times T \rightarrow \mathbb{N}_0$ *are called backward and forward* incidence functions*, respectively,*
4. $M_0 : P \rightarrow \mathbb{N}_0$ *is called* initial marking*.*

2.2.2 Colored Petri Nets

One extension is that of Colored Petri Nets (CPNs) [BK02]. Tokens receive a color property and can now be distinguished. The initial marking and incidence functions are now defined for a specific color. The different possibilities of firing a transition are referred to as modes.

2.2.3 Stochastic Petri Nets

Another extension is that of Generalized Stochastic Petri Nets (GSPNs) [BK02]. They introduce timed transitions that do not fire immediately like immediate transitions, but following an exponential firing delay distribution. Furthermore, firing weights can now be assigned to transitions to influence the probability of which transition is chosen among ready immediate transitions.

2.2.4 Queueing Petri Nets

Queueing Petri Nets (QPNs) build on a combination of CPNs and GSPNs, called Colored Generalized Stochastic Petri Nets (CGSPNs) [BK02]. A new type of place is introduced: that of a queueing place. A queueing place consists of a queue, a server and a depository. The server processes the tokens in the queue according to a certain scheduling strategy. The time a token occupies the server is defined through a statistical distribution. Once a token is finished, it is put into the depository, which then behaves like an immediate place for connected transitions. Only tokens in the depository are considered available for the incidence function. A formal definition [BK02] is given below:

Definition 2 *A Queueing Petri Net (QPN) is an 8-tuple* $QPN = (P, T, C, I^-, I^+, M_0, Q, W)$ *where:*

1. $P = \{p_1, p_2, \ldots, p_n\}$ *is a finite and non-empty set of* places*,*
2. $T = \{t_1, t_2, \ldots, t_m\}$ *is a finite and non-empty set of* transitions*,* $P \cap T = \emptyset$*,*
3. C *is a* color function *that assigns a finite and non-empty set of* colors *to each place and a finite and non-empty set of* modes *to each transition.*
4. I^- *and* I^+ *are the backward and forward* incidence functions *defined on* $P \times T$*, such that* $I^-(p,t), I^+(p,t) \in [C(t) \rightarrow C(p)_{MS}], \forall (p,t) \in P \times T$[1]
5. M_0 *is a function defined on* P *describing the* initial marking *such that* $M_0(p) \in C(p)_{MS}$.
6. $Q = (\tilde{Q}_1, \tilde{Q}_2, (q_1, \ldots, q_{|P|}))$ *where*
   - $\tilde{Q}_1 \subseteq P$ *is the set of timed queueing places,*
   - $\tilde{Q}_2 \subseteq P$ *is the set of immediate queueing places,* $\tilde{Q}_1 \cap \tilde{Q}_2 = \emptyset$ *and*
   - q_i *denotes the description of a queue taking all colors of* $C(p_i)$ *into consideration, if* p_i *is a queueing place* *__or__* *equals the keyword 'null', if* p_i *is an ordinary place.*
7. $W = (\tilde{W}_1, \tilde{W}_2, (w_1, \ldots, w_{|T|}))$ *where*
   - $\tilde{W}_1 \subseteq T$ *is the set of timed transitions,*
   - $\tilde{W}_2 \subseteq T$ *is the set of immediate transitions,* $\tilde{W}_1 \cap \tilde{W}_2 = \emptyset$ *and*
   - $w_i \in [C(t_i) \rightarrow \mathbb{R}^+]$ *such that* $\forall c \in C(t_i)$: $w_i(c) \in \mathbb{R}^+$ *is interpreted as a rate of a negative exponential distribution specifying the firing delay due to color c, if* $t_i \in \tilde{W}_1$ *__or__* *a firing weight specifying the relative firing frequency due to color c, if* $t_i \in \tilde{W}_2$.

2.2.5 Queueing Petri net Modeling Environment (QPME)

QPME, the Queueing Petri net Modeling environment, is a performance modeling tool based on the Queueing Petri Net (QPN) modeling formalism [KD09]. The QPME tool consists of two parts: QPE, the Queuing Petri net Editor, and SimQPN, a Simulator for Queueing Petri Nets.

QPE supports all of the Queueing Petri Net formalism and additionally supports hierarchical places, as well as a QPME specific extension: departure disciplines. The SimQPN simulator supports most, but not all, features of QPE. Departure disciplines allow enabling priorities based on colors for tokens inside the depository of a queueing place.

Another extension to the SimQPN simulator became available during the work for this thesis. In QPNs, token identities are not known. Only classes of tokens can be distinguished through token colors. To allow the computation of residence time distributions over an arbitrary subnet, individual tokens need to be tracked inside the corresponding subnet. The 'probes' feature allows to mark a set of colors, a starting place and an end place, between which tokens receive two additional properties: a timestamp and a probe id. The timestamp marks the time the token was generated at the start place, the probe id allows to distinguish timestamps from different probe definitions. When two or more tokens with timestamps are consumed by a transition, both the ids and timestamps would have to be matched. This, however, would prevent tokens from being consumed, that would otherwise be consumed without the extension. To conserve the original semantics of QPNs, timestamps are randomly chosen among marked tokens consumed by a transition.

[1]The subscript MS denotes multisets. $C(p)_{MS}$ denotes the set of all finite multisets of $C(p)$.

2.3 Model-to-Model Transformations

2.3.1 Overview

One possible way to define a model in the context of software engineering is "a formal representation of entities and relationships in the real world with a certain correspondence for a certain purpose" [Sta73](translated). This means that a model is an abstraction of a real world entity which captures only the information needed for a certain purpose. Furthermore, for the model to be useful, relationships that hold true for the model should also hold true for the real world. Otherwise, our reasoning based on the model is flawed.

A transformation is the automatic generation of a target model from a source model, according to a transformation definition [KWB03]. A transformation is not created for a single model which is known beforehand, but for a possibly infinite number of models to be created in the future. A transformation thus cannot be created based on any single model instance, but needs a more generic frame of reference. The reference we need is called a meta-model. It describes concepts that can be used for modeling the model [SV06]. The meta-model is what relates all the models we want to be able to apply the transformation to. It describes the space of relationships which can hold for model entities.

A transformation which operates on model instances that are well-defined by their corresponding meta-models is called a model-to-model transformation. If a transformation uses information outside of the source meta-model we speak of a parameterized transformation. The design space of model-to-model transformations is quite large and will not be described in detail here. More information can be found in [CH03].

Instead, we introduce some popular tools and standards in this area. Concerning meta-models, both the OMG standard MOF (Meta-Object Facility) [OMG06a] and EMF Ecore [emf], which is an implementation of Essential MOF (EMOF), are of great importance. In this diploma thesis all used meta-models are formalized in EMF Ecore.

While transformations can be, and still often are, written using a general-purpose language like Java, a number of domain specific languages for the area of model-to-model transformations are available. The transformation definition is interpreted by a transformation engine that executes the transformation on the input models. The OMG has published the QVT (Query View Transformation) [OMG08a] standard which incorporates a relational language (QVTR) as well as an operational language (QVTO). Another language is ATL, the ATLAS transformation language [JK06a]. Comparing the model-to-model transformation languages available, QVTR is purely declarative, QVTO is purely imperative and ATL is a hybrid which offers both imperative and declarative language elements [JK06b]. A case study comparing Java and ATL can be found in [CDGDM08].

QVTO is now covered in more detail as much of the implementation of the solver tool is realized with it.

2.3.2 QVT Operational

In this section, QVTO is introduced and simple examples are given for the most commonly used features. As this diploma thesis uses the QVTO implementation of the Eclipse project, the discussion is focused on that particular interpretation of the original standard [OMG08a]. Section 3.2.2 explains why QVTO was chosen. QVTO is an imperative language built on top of OCL [OMG06b]. The behavior of the mappings is defined in OCL, which is extended by several custom language features. The mapping and transformation declarations and related QVTO-specific features use a custom syntax. In our case, the QVTO scripts are run through a QVTO interpreter at transformation time. However, it is possible to integrate parts written in Java through so-called blackbox components.

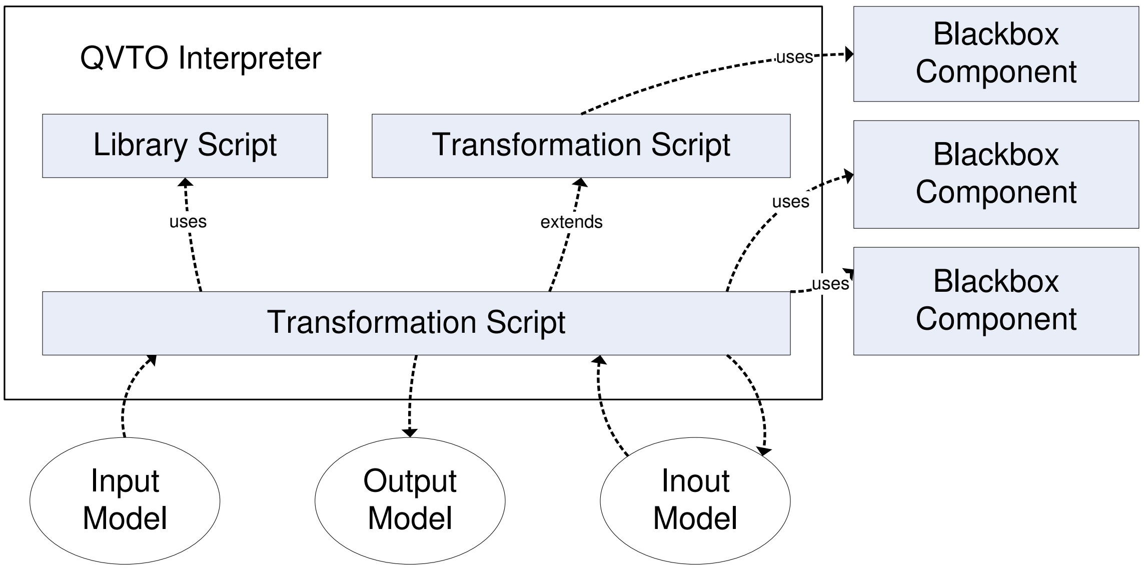


Figure 2.1: QVTO Architecture

Figure 2.1 shows the simplified architecture of QVTO. One QVTO script acts as the entry point to the transformation. It can use code imported from QVTO library scripts or extend the transformation defined in another transformation script. It can also use Java blackbox components. The import of a blackbox component looks just like the import of another script and the realization of imported helper methods is transparent to the host transformation.

Figure 2.2 shows the basic structure of a QVTO transformation script. Before the actual transformation is defined, imports and modeltypes are defined first. The imports point to other QVTO scripts or registered blackbox components. The modeltypes list the EMF packages, identified by the package namespace URI, that should be loaded into the execution context under a unique name. Apart from standard datatypes defined in QVTO, other entity types need to be loaded using a modeltype definition first. Otherwise their names cannot be resolved.

The transformation definition defines a name for the transformation and a number of input and output models. A model parameter can be declared as *in* for an input parameter, as *out* for an output parameter and as *inout* for a model that is transformed in place. Additionally, the library names that the transformation uses are listed.

The *main*() mapping always resembles the entry point to the (sequential) transformation steps. In the figure we show how the root object or root objects of in and inout parameter models can be retrieved and how a mapping is executed for them. Executing a mapping (with *map*) on a collection executes the mapping on each of the elements.

The other parts of the transformation script are represented through placeholders. Transformation properties can be defined that are valid throughout the whole transformation script. They are either configuration properties, which are set from the outside when executing the script, or they are intermediate properties which are managed from within the mapping. Figure 2.3 and Figure 2.4 show examples for both types of properties.

Figure 2.5 shows a simple example of a mapping definition. A mapping is defined for a model type (in this case A), has a name ($AtoB$) and a return type (B). It generally consists of a number of execution steps separated by ‘;’. Compared to a helper, which is discussed next, a mapping has special semantics. A mapping is only executed once for each matching element. Also, the result of a mapping can be accessed with trace functions.

```
import <QVTO script name without .qvto>;
import <Java blackbox namespace>.<java blackbox name>;
<...>

modeltype A uses "<EMF meta-model project package A namespace URI>";
modeltype B uses "<EMF meta-model project package B namespace URI>";
modeltype C uses "<EMF meta-model project package C namespace URI>";
modeltype D uses "<EMF meta-model project package D namespace URI>";
<...>

transformation <transf. name>(in a : A, out b : B, inout c : C <,...>)
        <optional> access library LibA LibB LibC <...> </optional>;

<configuration properties>
<intermediate properties>

// <comment>
// entry point to transformation
main() {
   a.rootObjects()[<classname of root objects>]-> map <mapping name>();
   c.rootObjects()![<classname of root object>].map <mapping name>();
   <...>
}

<mapping definitions>
<helper definitons>
```

Figure 2.2: QVTO Transformation Basic Structure

```
configuration property userPath : String;
```

Figure 2.3: QVTO Configuration Property Example

```
intermediate property <transf. name>::allNames : Set(String);
```

Figure 2.4: QVTO Intermediate Property Example

```
mapping A::AtoB() : B {
   name := self.name;
}
```

Figure 2.5: QVTO Mapping Example

There are many more variations of a mapping, however, they are out of the scope of this thesis.

```
helper setName(a : A, name : String) {
   a.name := name;

   return;
}

query nameOf(a : A) : String {
   return a.name;
}
```

Figure 2.6: QVTO Helper Example

Figure 2.6 shows the two types of helpers that can be defined. A helper behaves much like a Java method. Helpers defined using *helper* are expected to change the given parameters or cause other side effects. A *query* is expected to be free of side effects. Contrary to a *mapping*, helpers are always executed when they are called and no tracing information is generated. Like in Java, the *return* keyword is used to return the control flow from the helper and to set the result. *return* is not available in mappings. In a *mapping* an element of the result type is implicitly instantiated unless a special syntax is used. Again a detailed discussion is out of the scope of this thesis.

```
<imports>
<modeltypes>

library <library name>
   <optional> access library LibA LibB LibC <...> </optional>;

<helper definitions>
```

Figure 2.7: QVTO Library Structure

To finish the overview of QVTO, we will look at how the libraries that can be imported by a transformation are defined. Figure 2.7 shows a definition using QVTO. It looks much like a transformation definition. The differences are that the *library* keyword is used and that there are no parameter definitions.

To write a library using a Java class, we first need to register the target class under a name which is then used for the import statement in the QVTO script. This is done by registering an extension point in the plugin.xml of the host Eclipse plugin. Figure 2.8 shows the basic structure of the extension point definition. The blackbox component is put into a namespace and given a unique name. It is then found using

```
import <blackbox namespace>.<blackbox name>;
```

In addition to the target Java class (fully qualified by its package) the imported modeltypes need to be declared like in a QVTO script. The Java imports of the target class are not sufficient.

As we can see in Figure 2.8, a Java blackbox class looks just like a regular class. The only difference is that the exported methods carry the @*Operational* annotation. In our example, a global helper will become available as if

```
<extension point="org.eclipse.m2m.qvt.oml.javaBlackboxUnits">
   <unit name="<blackbox name>" namespace="<blackbox namespace">
   <library name="<blackbox name>" class="<java class of blackbox>">
      <metamodel
         nsURI="<namespace URI of package A">
      </metamodel>
      <metamodel
         nsURI="<namespace URI of package B">
      </metamodel>
      <...>
   </library>
   </unit>
</extension>
```

Figure 2.8: Plugin.xml Java Blackbox Definition

```
@Operation
public B globalHelper(A param) {
   <...>
}

@Operation(contextual = true)
public void workOnA(A self, Integer param) {
   <...>
}
```

Figure 2.9: Java Blackbox Operation Annotation

```
helper globalHelper(param : A) : B;
```

had been defined. The *contextual* = *true* setting makes a helper local to the type of the first parameter. Our example represents

```
helper A::workOnA(param : Integer);
```

3. Approach

The central goal of this diploma thesis is to evaluate the usefulness of QPNs as a target analysis model for PCM. This chapter describes how this main goal was achieved. The main goal consists of three parts: the derivation of a formal PCM-to-QPN mapping (Section 3.1), the development of a solver tool implementing the mapping (Section 3.2), and the evaluation of the mapping (Section 3.3).

3.1 Mapping Derivation

A formal mapping was derived for a selected set of features covering the major parts of PCM. To derive the mapping the following questions were asked for each of the mapped features:

- How is the feature represented in the PCM meta-model and what are the contexts it can be used in? Examples of context constraints include: all branch transitions of a branch must be of the same type, loop iteration specifications must be of an integral type.
- What is the simulation-time semantic of the feature and how does a concrete instantiation of the feature look like in the context of a PCM instance? The semantic is part of the PCM meta-model documentation. In unclear cases the simulation run-time behavior of the SimuCom solver is taken as a reference.
- Which combination of QPN entities has the desired effect during the simulation? Which approximations are necessary and which limitations apply?

3.2 PCM Solver Tool Development

In this section, the most important requirements that drove the development of the PCM Solver tool are introduced. The technology that was used for the implementation is described. Finally, the development process and its relation to the derivation of the formal mapping are explained.

3.2.1 Requirements

The requirements were derived from the goal to evaluate the PCM-to-QPN mapping in practice (see Section 3.3). In addition to the requirements, the approach taken to satisfy each requirement is also provided:

- It should be possible to configure the PCM Solver tool easily regarding all parameters that are to be varied for the evaluation (e.g., source model locations, measured metrics, maximum run time). This was achieved by providing a custom launch configuration type and user interface similar to the SimuCom configuration. To tell the tool which metrics to measure for which PCM elements, the ProbeSpec metamodel (Section 5.3.3) was used.
- The result metrics should directly relate to the annotated PCM elements and they should be available in a way suitable for automatic processing (e.g., using R). A module was developed that aggregates results data from the SimQPN simulator back into the PCM domain and prints the result to the console and to an output file. The module presents the metrics together with the name and the id of the source element and of the defining context elements (e.g., *AssemblyContext* for a *PassiveResource*).
- Automated execution of a number of simulation runs for pre-defined configurations should be possible. By encapsulating the solver process inside a PCM workflow job it can easily be executed outside of the context of a launch configuration.
- The maintainability of the code should allow efficient changes and refactorings to the transformation implementation. Automated JUnit tests based on separate fixtures for each feature were developed and maintained alongside the transformation implementation. This ensured that changes do not break unrelated features.

3.2.2 Technology

The PCM Solver tool uses QVTO as the model-to-model transformation language for the implementation of the PCM-to-QPN mapping. The reasons are discussed in this section. Some parts are implemented in Java and are integrated through black-box extensions. The decision to use QVTO was made after an initial vertical prototype had successfully been developed.

The reason why a domain specific language was chosen over Java is that the transformation can be written more concisely and less verbose. A comparison of ATL and Java in [CDGDM08] shows that using a domain specific language has many benefits, including better maintainability. The prototype showed that the major drawbacks of increased initial startup effort and of tool limitations were not a major issue in this case. Current tool support is very mature and features like content assist and syntax highlighting are available for QVTO, QVTR and ATL.

Among the transformation languages, QVTO and ATL looked most promising as they offer imperative constructs. A pure declarative description using QVTR is difficult, as no clear 1-to-1 relationship between PCM model elements and QPN elements exists. For example, a branch inside a SEFF is not transformed once but each time for the different assembly and usage contexts. An imperative approach that traverses the SEFF multiple times, using a different context each time appeared to be the better and more natural approach.

Between ATL and QVTO, QVTO was chosen because an integration component for the PCM Workflow Engine (see Section 6.1) already existed. This greatly reduced the effort of getting started with the prototype. The documentation of QVTO is very detailed in the form of the OMG specification [OMG08a]. Being based on a major standard, future adoption seems likely. One drawback compared to ATL is the lack of debugging support in the free version of the Eclipse QVTO plugin. In practice this turned out to be an acceptable limitation. Furthermore, future versions of the Eclipse QVTO implementation are likely to correct this.

The tool plugins are based on and developed using *Eclipse Galileo 3.5.1* with the following features installed:

- EMF 2.5.0
- Eclipse QVT Operational 2.01
- PCM 3.2 Development Build
- QPME 1.5.2 Development Build

3.2.3 Development Process

The PCM Solver tool was developed using an iterative process. This process ran in parallel to the formal mapping efforts so that insights during the implementation became available to further improve the formal mapping. This minimized the risk of developing in the wrong direction. Refactoring steps were executed in order to keep the code simple and to increase reusability. Test cases were created to focus the implementation efforts. For each PCM feature, at least one increment was passed through. If major variants were identified, more increments were necessary. Support for the ProbeSpec decorator model (Section 5.3.3) and the results aggregation module (Section 5.3.4) were added later in the thesis. Each feature increment followed these steps:

1. Create a feature test fixture.
2. Create the most important feature test cases.
3. Add the QVTO implementation to the transformation.
4. If necessary, try to update the DependencySolver module. If this is not possible or it would take too long, use dummy values.
5. Refactor the transformation and helper libraries.

Once the decorator model became available, each feature implementation was iteratively updated using the following steps:

1. Check if the feature needs instrumentation.
2. If it does, add a decorator model fixture to the test fixture.
3. Add to or update the test cases.
4. Update the transformation implementation.
5. Refactor the transformation and helper libraries.

Similarly, once the results integration and simulator integration were complete, the following steps were iteratively executed:

1. Choose a small number of features that fit together.
2. Create an integration test fixture that uses all those elements in a reasonable way.
3. Add a matching decorator model.
4. Write an integration test that checks the results of the defined metrics.
5. Fix any problems found with the mapping and implementation and update the corresponding test cases.
6. Refactor the transformation and helper libraries.

3.3 Evaluation

This section presents the approach towards the evaluation of the formal mapping and PCM Solver tool implementation. The evaluation is split into two parts. The evaluation of the support of the individual features considered for the mapping, and the case studies.

3.3.1 Feature Support

To evaluate the feature support, small models were analyzed using the following available solvers: SimuCom, SimQPN, LQSim and LQNS. For each feature a separate, but complete, model was created that uses a minimal set of other features. Simple metrics (like usage scenario throughput, processing resource utilization) or simple derived metrics (like the number of requests inside the system) were calculated manually. The results of the different solvers were then compared to those reference metrics.

3.3.2 Case Studies

The purpose of the case studies was to evaluate the accuracy and analysis overhead of the available solvers on models of realistic size and complexity. As a single model is not enough for an in-depth evaluation, five case studies were conducted with models from external sources. Only models that could be migrated to the employed PCM version were used. In cases a model used stochastic expressions that were not supported by the DependencySolver, those expressions were either removed, or the scenario was discarded from the evaluation. The predictions of the SimQPN, LQSim and LQNS solvers were compared to the predictions of the SimuCom solver, which served as the reference solver for the evaluation.

4. Related Work

This chapter presents the related work of this thesis. Most directly related are other solvers available for PCM. They are discussed in Section 4.1. There are many alternatives to PCM for performance prediction. Relevant to this thesis are other model-based approaches that naturally use some form of transformation to analyze the model instances. A broad overview is given in Section 4.2. Finally, in Section 4.3, a number of related intermediate languages in the domain of software performance engineering are discussed.

4.1 PCM Analysis Techniques

The techniques for analyzing PCM instances can be separated into techniques that fold stochastic parameter expressions down to a single distribution and those that support stochastic expressions at the analysis level. The DependencySolver, a module that resolves parametric dependencies, has been introduced in [Koz08, pp. 189]. It is used by the transformation in this thesis, by the transformations to Layered Queueing Networks (LQNs) [KR08] and by the transformation to Stochastic Regular Expressions [Koz08, pp. 199].

The transformation to LQNs is closely related. However, LQNs have different expressive power than QPNs. A detailed comparison is not part of this diploma thesis and can be found in [Hei07]. Stochastic Regular Expressions are solved analytically instead of through simulation. The analysis is generally much faster but only single user scenarios can be evaluated and the level of detail of the results is very limited.

Other techniques based on PCM are a transformation to an extended form of QPNs which has not been studied by the research community and is not supported by available tools [Koz08, pp. 141]. The most important difference is the use of tokens that carry arbitrary properties instead of just a color value.

[Hen10] proposes a PCM transformation to OMNeT++, a simulation framework which includes a language for the definition of the network topology. A decorator model is employed to generate a much more realistic network infrastructure closer to the OSI reference network model. However, an experimental evaluation of the approach has not yet been published.

Finally, the PCM-Bench tool comes with the SimuCom simulator [BKR09]. The PCM instance is run through a model-to-text transformation. Java code is generated that builds on Desmo-J, a general simulation framework. The code is then compiled on-the-fly and executed. SimuCom is tailored to support all of the PCM features directly and covers the whole PCM meta-model. Limitations with the employed version include a limited set of results metrics. It lacks configurability regarding those metrics. Furthermore, the scalability, especially due to the high analysis overhead and due to memory constraints is limited. Other techniques have the potential to execute faster without sacrificing much results accuracy.

4.2 Model-Based Approaches to Performance Prediction

The biggest group of model-based approaches to performance prediction are based on the Unified Modeling Language (UML) [OMG07], the most commonly used general-purpose software meta-model. A distinction can be made depending on whether or not an annotation profile is used. The most important standards for performance annotations in UML are SPT (UML Profile for Schedulability, Performance and Time) [OMG05] and the newer and extended version MARTE (UML Profile for Modeling and Analysis of Real-time and Embedded Systems) [OMG08b]. While the MARTE profile allows for a detailed performance specification in many software engineering domains, it suffers from the same drawbacks that UML suffers from. Because the meta-model is very large and the semantics are often ambiguous, it is unpractical to create an analysis tool that covers all of the profile's features. For this reason current approaches only support a limited number of MARTE or SPT stereotypes.

We start with approaches that make use of standardized performance profile annotations: Di Marco and Inverardi [MI04] transform UML models annotated with SPT stereotypes into a multichain queueing network. UML-ψ, the UML Performance SImulator [MB04], comes with its own simulation model. A UML instance annotated with SPT stereotypes is transformed to this model. As the simulation model is close to UML, the results can easily be reported back to the annotated UML instance [MM06]. Another approach uses the stochastic process algebra PEPA as analysis model [TG08]. In this case, only UML activity diagrams are considered, which are annotated with a subset of the MARTE stereotypes. A software tool implementing this method is also available. [BM04] integrate their approach into the Argo-UML modeling tool, using the RT-UML performance annotation profile. An execution graph and a queueing network serve as the target analysis formalisms.

In the following, we present approaches that use UML, but do not use standardized performance profile annotations: Petriu et al. [GP02] use XSLT, the eXtensible Stylesheet Language Transformations, to execute a graph pattern based transformation from a UML instance to LQNs. Instead of annotating the UML model, it has to be modeled in a way so that the transformation can identify the correct patterns in the model. Bernardi et a. [BDM02] consider only UML statecharts and sequence diagrams and assume that those two diagram types cover all relevant aspects of the system. A transformation written in Java turns the model into GSPN submodels that are then combined into a final GSPN. [GM01] use UML with custom XML performance annotation. The performance model is not described in detail, but appears to be based on queueing networks. [WW04] use UML component models together with a custom XML component performance specification language. LQN solvers are used for the analysis.

Other approaches exist that are not based on UML: [BMdW$^+$04] and [BdWCM05] build on the ROBOCOP component model. For the analysis a proprietary simulation framework is used. [EFH04] propose a custom control flow graph model notation and custom simulation framework. [HMSW02] employ the COMTEK component technology, coupled with a proprietary analysis framework. [SKK$^+$01] specify component composition and performance characteristics using a variant of the big-O notation. The runtime analysis is not discussed in detail.

An elaborate comparison of the discussed and other performance evaluation approaches for component-based software systems can be found in [BGMO06], and more recently, in [Koz09].

More information on transformations in the software performance engineering domain, especially about some of the UML-based approaches and the intermediate languages can be found in [MM06].

It is noteworthy that Queueing Networks (QNs) and Layered Queueing Networks (LQNs) are not the only common analysis formalisms. Markov chains, Stochastic Petri Nets and Stochastic Process Algebras are also used for performance analysis [BKR09].

4.3 Intermediate Software Performance Languages

To bridge the gap between the software engineering and the performance analysis domains, a number of intermediate languages (or kernel languages) for specifying software performance information have been published. They are not used for this approach because of the increased complexity they would introduce and their lack of adoption. Still they are related as they aim at reducing the effort of creating transformations in the software performance engineering domain.

The idea is that instead of having to develop $M \cdot N$ transformations for M source meta-models and N target meta-models, only $M + N$ transformations are needed. M transformations to turn the source model into an instance of the intermediate language and N transformations to turn the intermediate language instance into the target model [MM06].

In [SLC$^+$05] an intermediate language called SPE Meta-Model and a corresponding XML-based interchange format called S-PMIF are introduced. S-PMIF is meant to facilitate the transfer of information between software design tools and performance analysis tools. In a proof-of-concept, the new format is used in conjunction with SPE·ED. SPE·ED is a commercial software performance modeling tool.

KLAPER (Kernel LAnguage for PErformance and Reliability analysis) [GMS07] is a meta-model defined in MOF. It is designed to capture the relevant information for the analysis of non-functional attributes of component-based systems. Mappings depicted using QVTR diagrams from KLAPER to both discrete time markov processes (DTMPs) and extended queueing networks (EQNs) are also provided.

CSM, the Core Scenario Model [PW07], is a meta-model that captures the essential entities in the SPT Profile domain model which are required for building performance models. However, even though it was derived in the context of using UML and SPT, it is meant to function as a general intermediate language for software performance evaluation.

While these intermediate meta-models differ in detail, they share the main concepts used in software performance engineering: That of resources and of behavior that references and uses those resources. Like in PCM, the behavior is formalized as some form of abstract control flow.

5. PCM-to-QPN Mapping

This chapter describes the PCM-to-QPN mapping from a mostly theoretical point of view. Section 5.1 introduces the DependencySolver and how it simplifies the PCM instance for the remainder of the mapping. Section 5.2 describes in detail the mapping of the PCM features covered in this thesis to QPNs. Finally, Section 5.3 deals with the metrics that are supported by the PCM Solver Tool (Chapter 6). It is shown how the generated QPN is extended and how the collection of data is configured so that a simulation run provides all needed information.

5.1 Dependency Solver Preprocessing

5.1.1 Overview

The DependencySolver is a module created by Heiko Koziolek as part of his PhD thesis [Koz08, pp. 189]. It is a tool for substituting parameter names inside PCM stochastic expressions with characterizations originating from the usage model. In addition, it also handles component parameters. PCM stochastic expressions are briefly introduced in Section 2.1.6.

Figure 5.1 illustrates the basic idea behind the DepdendencySolver. The performance of a component depends on its context. The component authors can not foresee all possible use cases and therefore they use stochastic variables inside the behavioral constructs and resource demands. The domain expert later specifies concrete values for these parameters. The main task of the DependencySolver is to resolve these parametric dependencies as a preprocessing step. It replaces variable references with their concrete stochastic expressions and reduces the resulting expression to either a single number literal or to a single probabilistic function literal. Discrete expressions are reduced to a probabilistic mass function, continuous expressions to a probabilistic density function. As the values of the stochastic variables also depend on the system composition and system allocation, the DependencySolver manages these contexts as well during traversal of a PCM instance. A more detailed description of the system composition and how the composition is traversed can be found in Section 5.2.3.

Using the DependencySolver comes with a number of benefits. The subsequent mapping steps are made less complicated as all stochastic expressions are resolved and free of stochastic variables. Also there is no need in this thesis to write complicated context management code. The drawback is that the DependencySolver needs to be maintained and

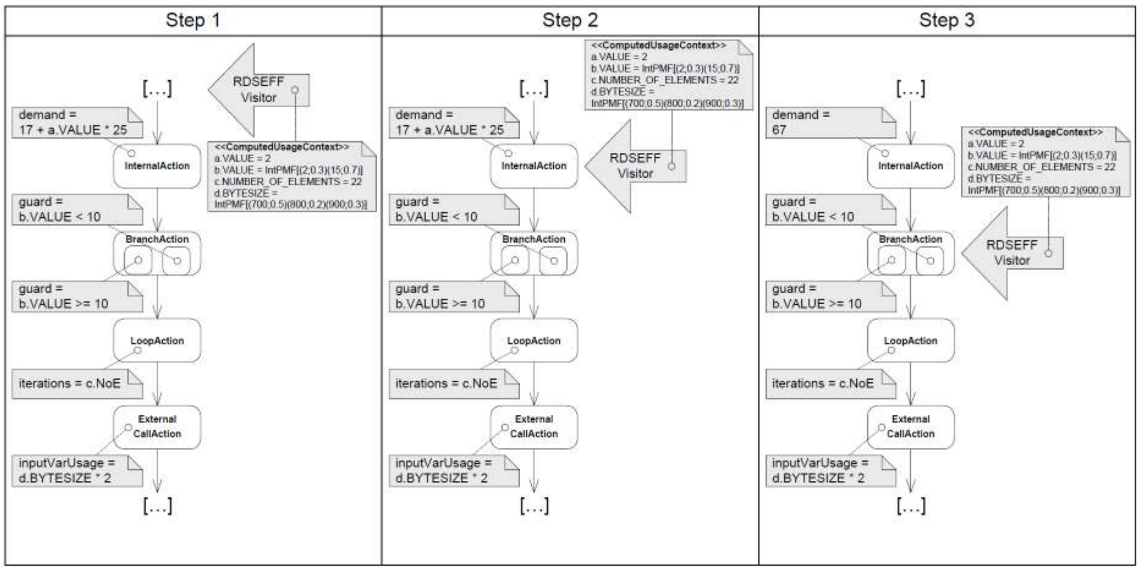


Figure 5.1: DependencySolver Illustration [Koz08, p. 194]

extended during the work for this thesis. For example, required extensions are the support for *ForkAction*s, *ComponentSpecifiedExecutionTime*s and *ExternalCallAction* parameter bytesizes.

As an alternative to using the DependencySolver it was considered to write a context management module similar to the context management system of the SimuCom simulator. It is inspired by compiler technologies and involves stack frames [BKR09]. Stochastic expressions are reduced on-the-fly during traversal. The module implementing the reduction of expressions and the folding of distributions could have been reused.

The decision was made to reuse the DependencySolver as it was not clear how much effort the implementation of a custom context handling module would have required and it was easy to integrate the DependencySolver into the prototype.

5.1.2 Limitations

The DependencySolver introduces a number of limitations: The dependencies between stochastic variables are not considered. This can cause inaccurate results (e.g., when evaluating nested conditional branches that depend on a single stochastic variable). More information can be found in Section 5.2.4. Not all boolean expressions can be evaluated. The AND operation is not supported. This is a non-trivial problem as a solution would require a model of the stochastic dependencies of distributions to compute dependant probabilities and is out of the scope of this thesis. All information about stochastic variables is lost, and with it the variable scopes. One example of a variable scope in PCM is a *Service-EffectSpecification* (SEFF). The value which is determined for one of the input parameters holds for all expressions that use that variable inside the SEFF until it is overridden by a return parameter usage or the SEFF ends. We do not know if the expression for the *ResourceDemand* of an *InternalAction* once contained a reference to a variable which was overwritten with the return value of an *ExternalCallAction*. Each time an expression is used, it is evaluated again and independently of other evaluations.

These limitations go beyond implementation issues and require further research. They are inherent in mapping a semantic in which requests have an id and a state to a semantic which only supports request classes. Consequently, these issues would arise even without the use of the DependencySolver.

In addition, implementation issues were identified with the module implementing the folding and reduction of stochastic expressions. Number arithmetic like ‘3 * 4 + 5’ = ‘17’ generally does not cause any problems. Neither do simple expressions involving operations on distributions like ‘3 * IntPMF[(2;0.2)(3;0.8)]’ = ‘IntPMF[(6;0.2)(9;0.8)]’. Other expressions like multiplying a double constant with an IntPMF were not implemented in the employed version.

5.2 Feature Mappings

5.2.1 Overview

An introduction to the PCM meta-model can be found in Section 2.1. A more detailed introduction can be found in [Koz08, pp. 74]. In this section, a basic understanding of PCM and its larger components is assumed. For each feature only the relevant meta-model entities and relationships are shown.

For each feature group the following topics are covered:

- A short general description.
- Description of the related PCM meta-model entities and their relationships.
- Gathering of information required for the mapping.
- QPN mapping and in non-trivial cases a more detailed explanation of its semantics.
- Limitations regarding any of the features in the group.

To improve readability, PCM meta-model element names, feature names as well as QPN element names (of places and transitions) are *emphasized.*

As the mapping described in this section is implemented (Chapter 6) and evaluated (Chapter 7) using the SimQPN simulator [KD09], several limitations apply to the supported QPN elements available for the mapping:

- No timed transitions.
- No hierarchical (subnet) places.
- No priorities for transitions.
- For places:
  - No max, no ranking and no priority properties.
  - No PRIO and no RANDOM scheduling strategies.

The PCM diagrams follow the notation of UML class diagrams [OMG07]. To improve readability and to reduce clutter the following simplifications are made regarding the presentation of PCM concepts:

- The *id* and *name* properties of the common base entities *Identifier* and *Entity* are omitted from the diagrams.
- Entities and properties that are unrelated to the mappings described in this thesis have been removed from the diagrams. This includes all OCL constraints as well as all properties only used for reliability analysis.
- Simplified attribute and reference names of all entities are used. In the meta-model each property is postfixed with an underscore followed by the class name that introduces that feature.

- All comparisons between PCM entities are implicitly made by *id*.
- A section on *SetVariableActions* has been omitted as they contain no useful information after the DependencySolver has removed all references to stochastic variables. A simple connector transition is generated in their place.

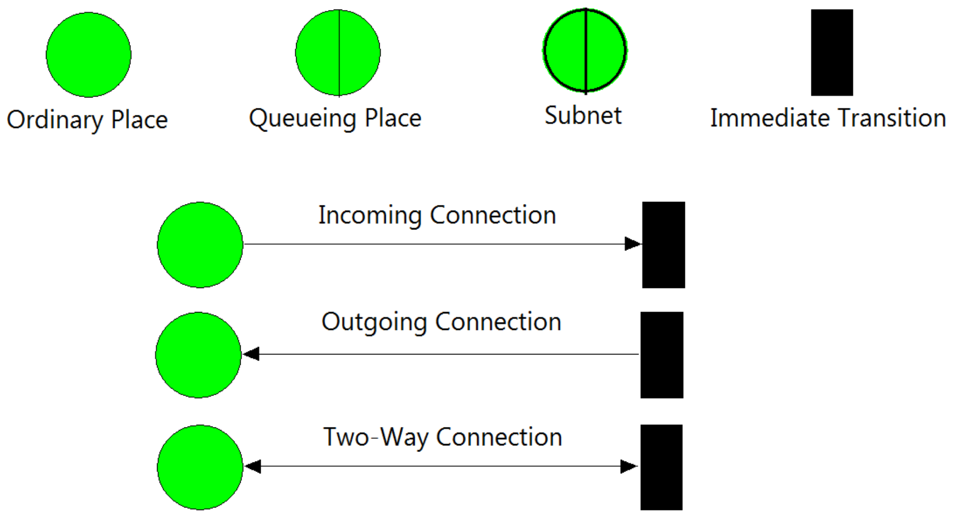


Figure 5.2: QPN Diagram Legend

The QPN diagrams employ the QPME editor [KD09] visualizations. A legend is presented in Figure 5.2. The following simplifications have been made for QPN concepts:

- Subnet places, though not available for the mapping, are used to represent parts of the QPN which vary independently of the described feature.
- QPN element names have been simplified to increase readability. In the actual mapping all elements are given a unique name generated from the host element *id*.
- In the following, the term transition means an immediate transition. Timed transitions are not available for the mapping.
- If a place contains only one color reference or in other obvious cases, the mapping details are not explicitly mentioned.
- The departure discipline of all places is set to NORMAL.

5.2.2 Workload

As we can see in Figure 5.3, each *UsageSenario* contains exactly one *Workload* and exactly one *ScenarioBehavior*. The *Workload* defines how often the defined *ScenarioBehavior* will be executed and can be either of type *ClosedWorkload* or of type *OpenWorkload*. An *OpenWorkload* is characterized by its *inter-arrival time* in seconds. After each time segment, a new user is created at the *Start* node of the *ScenarioBehavior*. A *ClosedWorkload*, on the other hand, defines a fixed number of users, the *population*, with a specified a *think time* in seconds. All users start in parallel. Once a user reaches the *Stop* node of the *ScenarioBehavior*, the user waits for the specified *think time* and is then released into the *ScenarioBehavior* again.

Figure 5.4 shows how the *OpenWorkload* is represented in the QPN. The *Client-Place* queueing place generates tokens of a color which is different for each *UsageScenario*. The name of the color is prefixed by the *UsageScenario* id. This allows to trace the time that requests spend on queues back to the originating *UsageScenario* (see Section 5.3). The

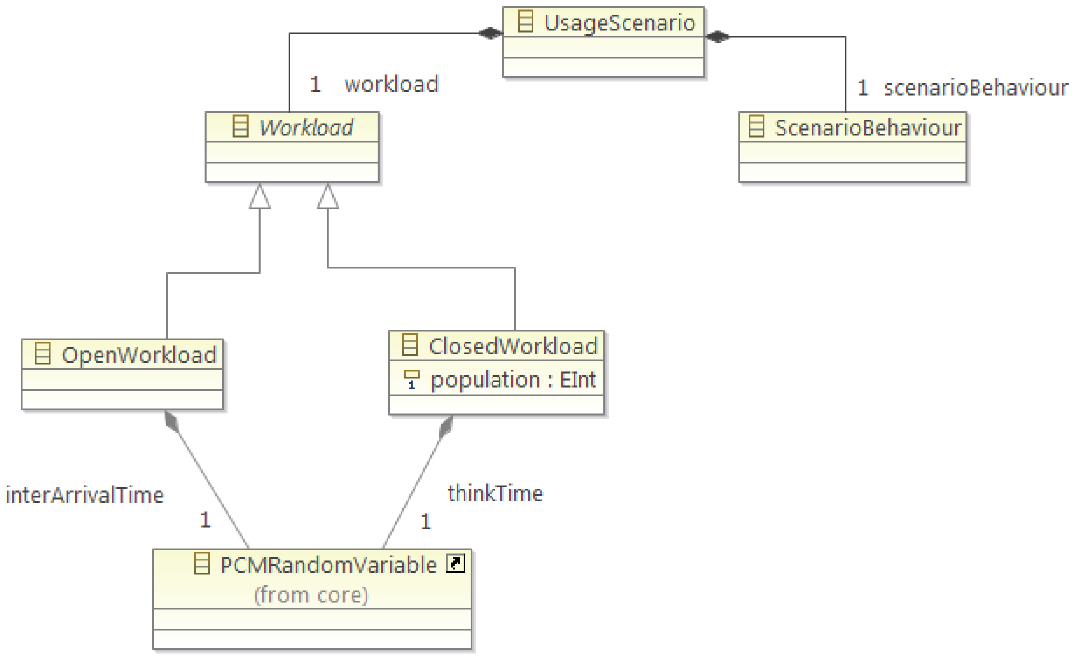


Figure 5.3: Workload PCM

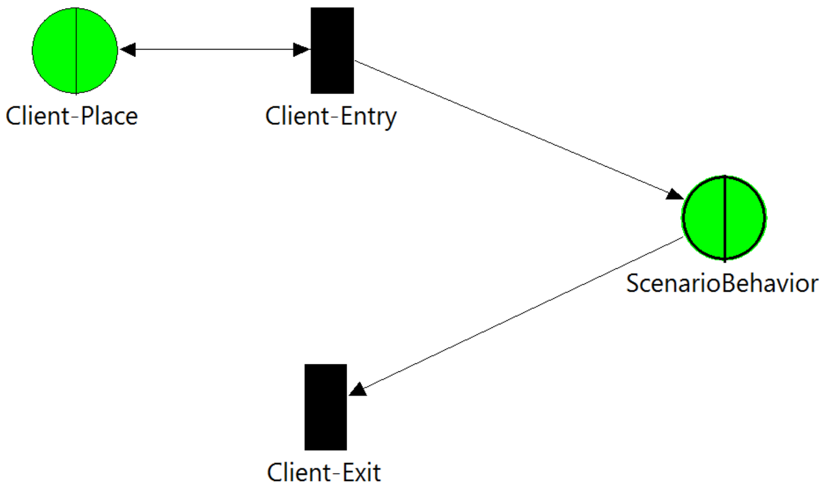


Figure 5.4: OpenWorkload QPN

Client-Place references a client queue, which is only used by this part of the mapping. Its scheduling strategy is set to Infinite Server (IS). An empirical distribution is used for the *Client-Place* resource demand equal to the *inter-arrival time* distribution of the *OpenWorkload*. The initial number of tokens at the *Client-Place* is set to 1. The *Client-Entry* transition is now configured in a way that for each input token it will create a new token in the first place of the QPN generated for the *ScenarioBehavior* and another token in the *Client-Place* queue. The *Client-Exit* transition consumes tokens of the color for this *UsageScenario* in the last place of the *ScenarioBehavior* subnet but does not create any new tokens.

When the *Client-Entry* transition creates a new token in the *ScenarioBehavior* subnet, the input token is immediately returned to the queue of the *Client-Place*. The next token will then become available after the residence time in the queue. As the queue has an Infinite Server (IS) scheduling strategy, there is no contention and thus the residence time equals the resource demand of *inter-arrival time*. This combination therefore leads to the expected behavior, that each *inter-arrival time* seconds a new token is generated.

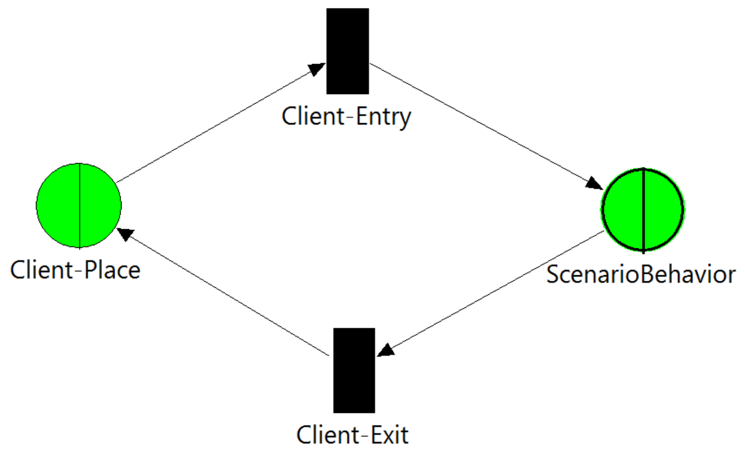


Figure 5.5: ClosedWorkload QPN

As we can see in Figure 5.5, the mapping for a *ClosedWorkload* is not much different. Again we have a *Client-Place* queueing place and the *Client-Entry* and *Client-Exit* transitions. The same type of client queue and color are used. The major difference is that the *Client-Entry* transition does not generate any tokens in the *Client-Place*. Instead, this is done by the *Client-Exit* transition. At the *Client-Place* we use an empirical distribution for the resource demand equal to the *think time* distribution of the *ClosedWorkload*. The initial number of tokens is set to its *population*.

A new token will now be available after it has passed through he whole *ScenarioBehavior* and after the residence time in the queue. Again there is no contention as the queue scheduling strategy is set to Infinite Server (IS). The residence time thus equals the resource demand of *think time*. As the initial number of tokens are immediately available to the *Client-Entry* transition and all our transitions are immediate transitions, we obtain the expected behavior of *population* tokens that enter the system in parallel and are delayed for *think time* seconds before entering the *ScenarioBehavior* again.

Formally there are no limitations for this part of the mapping. One limitation concerning the current solver tool implementation is that the *inter-arrival time* and the *think time* are reduced from the distribution computed by the DependencySolver to a single mean value. A deterministic distribution is used instead. The reason is that in the employed version of SimQPN, the support for empirical distributions does not meet the requirements of this mapping. This will be fixed in upcoming versions.

5.2.3 Calls

In the PCM meta-model there are two types of call entities. The *EntryLevelSystemCall* and the *ExternalCallAction*. They both represent the invocation of a single method of an interface that is offered by one of the components in the repository. While the resulting QPN mapping constructs are very simple, obtaining the necessary information to complete the mapping is far from trivial. The situation is complex because the PCM model contains the information about the component behavior and the component assembly in different locations that can be referenced from multiple other locations. Even though the DependencySolver module is used to manage the related information, we will cover the information flow and retrieval in detail because it is central to understanding the PCM model and the mapping as a whole. The situation is less complicated for *EntryLevelSystemCall*s, thus they are discussed first.

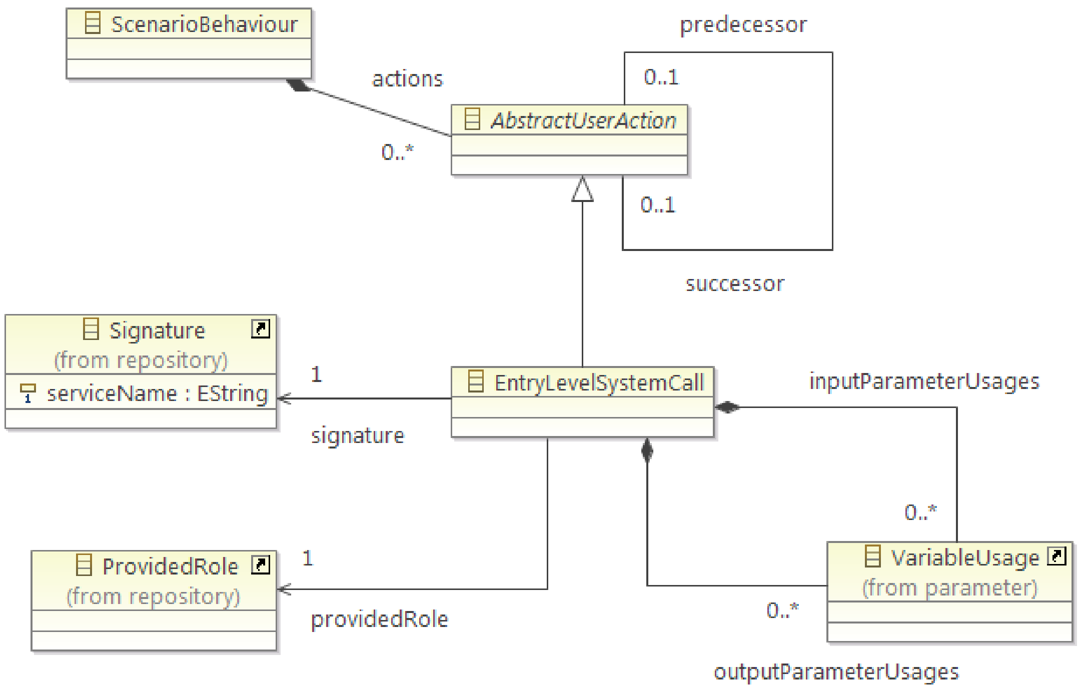


Figure 5.6: EntryLevelSystemCall PCM

Figure 5.6 shows the *EntryLevelSystemCall* and related entities. The available information can be grouped into two parts. One part is made up by the two *VariableUsage* references *inputParameterUsages* and *outputParameterUsages*. This part is only used when stochastic variables come into play and is not crucial for the following discussion. It is consumed exclusively by the DependencySolver (Section 5.1) and will not be covered here in more detail. The important part is made up of the *ProvidedRole* and *Signature* references. *BasicComponent* is the only component type that contains concrete behavioral information in the form of a *ServiceEffectSpecification* (SEFF). The *BasicComponent* contains a SEFF for every *Signature* of every *Interface* it provides. Once the right *BasicComponent* has been identified, the *signature* of the *EntryLevelSystemCall* uniquely identifies the SEFF (the *Signature* names are assumed to be unique across all provided *Interfaces*). Multiple components can provide the same *Interface*. Furthermore, a component can be used in multiple locations in any *ComposedStructure* (*System*, *CompositeComponent* and *SubSystem*). This indirection is implemented by the use of *AssemblyContexts*, which are separate entities used for the assembly and which reference the target component contained in one of the repositories.

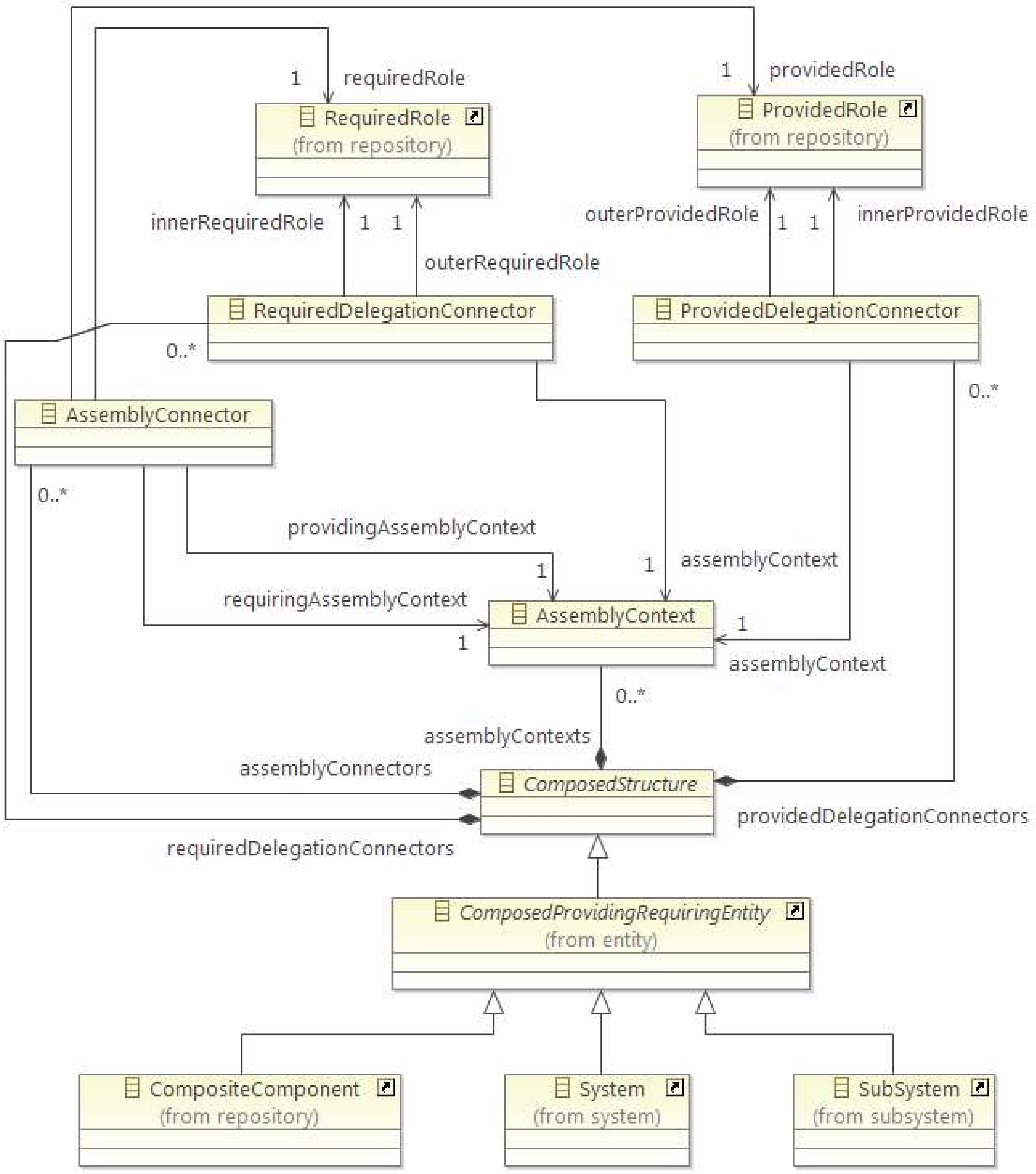


Figure 5.7: ComposedStructure PCM

To find the correct behavior to include in the generated QPN, we must navigate the composition hierarchy of the PCM instance. In case of an *EntryLevelSystemCall*, we always start at the single *System* instance which is at the top of the composition hierarchy. Apart from being the entry point of our navigation, the *System* behaves like any other *ComposedStructure*. Figure 5.7 shows most of the entities that make up the composition structure. Like a *BasicComponent*, each *ComposedStructure* has a list of provided and another list of required *Interfaces*. They share the common base entity *InterfaceProvidingRequiringEntity*. However, instead of specifying the behavior directly, *AssemblyContexts* are used that point to another *InterfaceProvidingRequiringEntity*. The *Interfaces* are also not referenced directly, but through a *Role* of which there are two concrete types. A *ProvidedRole* and a *RequiredRole*. The pair of *Role* and *Interface* is analog to the pair of *AssemblyContext* and

RepositoryComponent. The contained *AssemblyContexts* are now connected to the outside using *ProvidingDelegationConnectors* and *RequiredDelegationConnectors*, determining to which *AssemblyContext* a request will be routed when entering the *ComposedStructure*. An *AssemblyConnector* can further be used to connect the *RequiredRole* of the *RepositoryComponent* of one *AssemblyContext* to the *ProvidedRole* of the *RepositoryComponent* of another *AssemblyContext* located in the same *ComposedStructure*. Figure 5.8 shows the missing pieces of the composition hierarchy, including the relation between *AssemblyContext* and *RepositoryComponent*.

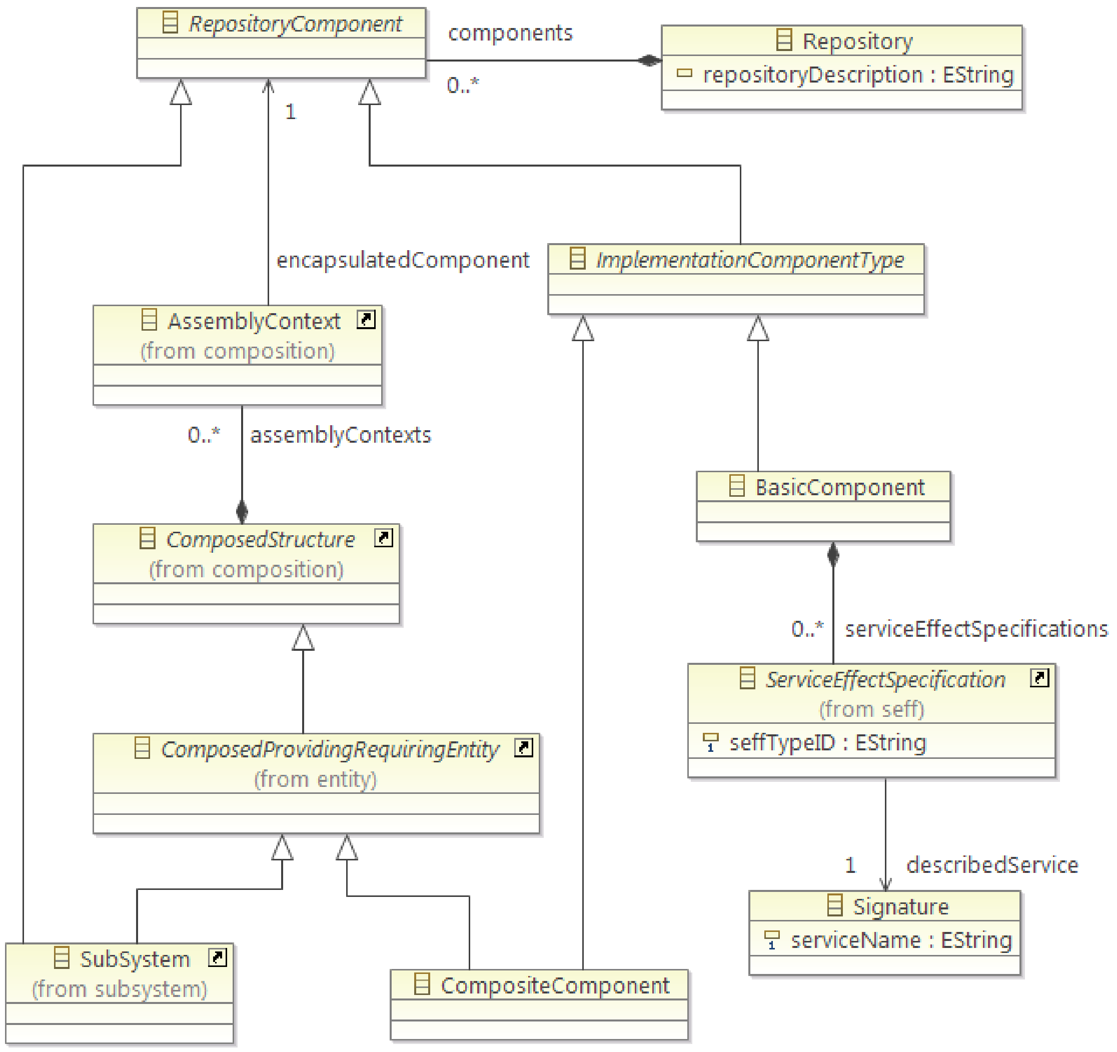


Figure 5.8: RepositoryComponent PCM

To find the correct *BasicComponent* at the end of the composition hierarchy, given the entry *ProvidedRole* P of a *ComposedStructure* S, the following steps are followed:

1. Select from $S.providedDelegationConnectors$ the *ProvidedDelegationConnector* C whose *outerProvidedRole* matches P.
2. If $C.assemblyContext.encapsulatedComponent$ is of type *BasicComponent*, return it.
3. If $C.assemblyContext.encapsulatedComponent$ is of type *ComposedStructure* return to step 1 with the component as S and $C.innerProvidedRole$ as the new P.

After retrieving the target SEFF of the *EntryLevelSystemCall*, the mapping can continue. Figure 5.9 shows the result. The *EntryLevelSystemCall* is represented by two transitions.

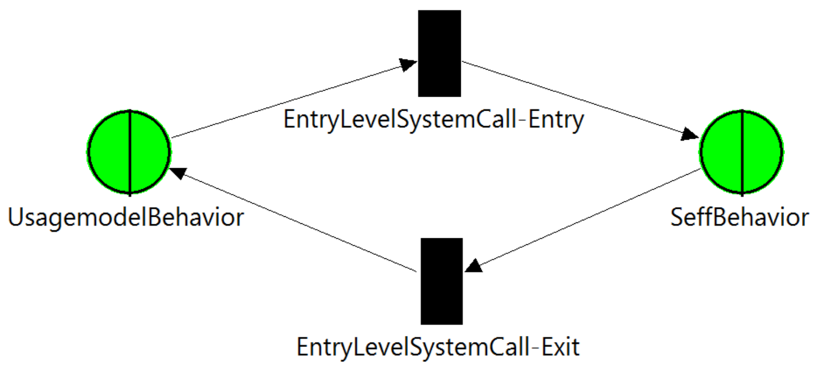


Figure 5.9: EntryLevelSystemCall QPN

EntryLevelSystemCall-Entry connects the last place of the mapping of the previous usage model *AbstractUserAction* with the first place of the mapping for the target SEFF. *EntryLevelSystemCall-Exit* in return connects the end of the SEFF mapping with the beginning of the following *AbstractUserAction*. The submodels not directly part of this mapping are abbreviated by the subnet places *UsagemodelBehavior* and *SeffBehavior*. It is important to note that a new token color is created for each *EntryLevelSystemCall* that distinguishes the requests coming from that call. This is important as the places and transitions of a SEFF are reused much like a *BasicComponent* is reused in a PCM instance. Even two *EntryLevelSystemCall*s that share the same *ProvidedRole* and *Signature* can differ in their *VariableUsage* and thus result in different resource demands at an *InternalAction*.

Looking at Figure 5.10 we see that for an *ExternalCallAction* we start with almost the same information as when we start with an *EntryLevelSystemCall*. We have both a *Signature* and a *RequiredRole* reference and *VariableUsage* information that is handled by the DependencySolver. Locating the target *BasicComponent* is, however, much more difficult compared to the case of an *EntryLevelSystemCall*. The reason is that we do not always start at the outside of the top-most *ComposedStructure*, but at an *AssemblyContext* of any of the *ComposedStructure*s that are directly or indirectly referenced from the usage model. In addition to the current *AssemblyContext* we will also need information about parent *AssemblyContext*s further up in the composition hierarchy that were visited during the traversal of the PCM instance up to the current point. In the following, this information is assumed to be available. To find the correct *BasicComponent* that this *ExternalCallAction* points to given the current composition context and the entry *RequiredRole* R, the following steps are executed:

1. Let S be the parent *ComposedStructure* of the current *AssemblyContext* X.

2. Select from *S.requiredDelegationConnectors* the *RequiredDelegationConnector* D whose *innerRequiredRole* matches R and whose *assemblyContext* matches X.

3. If we find a match, set X to the parent *AssemblyContext* of X. If there is no parent, we have reached the *System* and we have to deal with a system required interface (Section 5.2.10). Return to step 1 with *D.outerRequiredRole* as the new R.

4. If we do not find a matching *RequiredDelegationConnector*, select from *S.assemblyConnectors* the *AssemblyConnector* A whose *requiredRole* matches R and whose *requiringAssemblyContext* matches X.

5. If *A.providingAssemblyContext.encapsulatedComponent* is of type *BasicComponent*, return it.

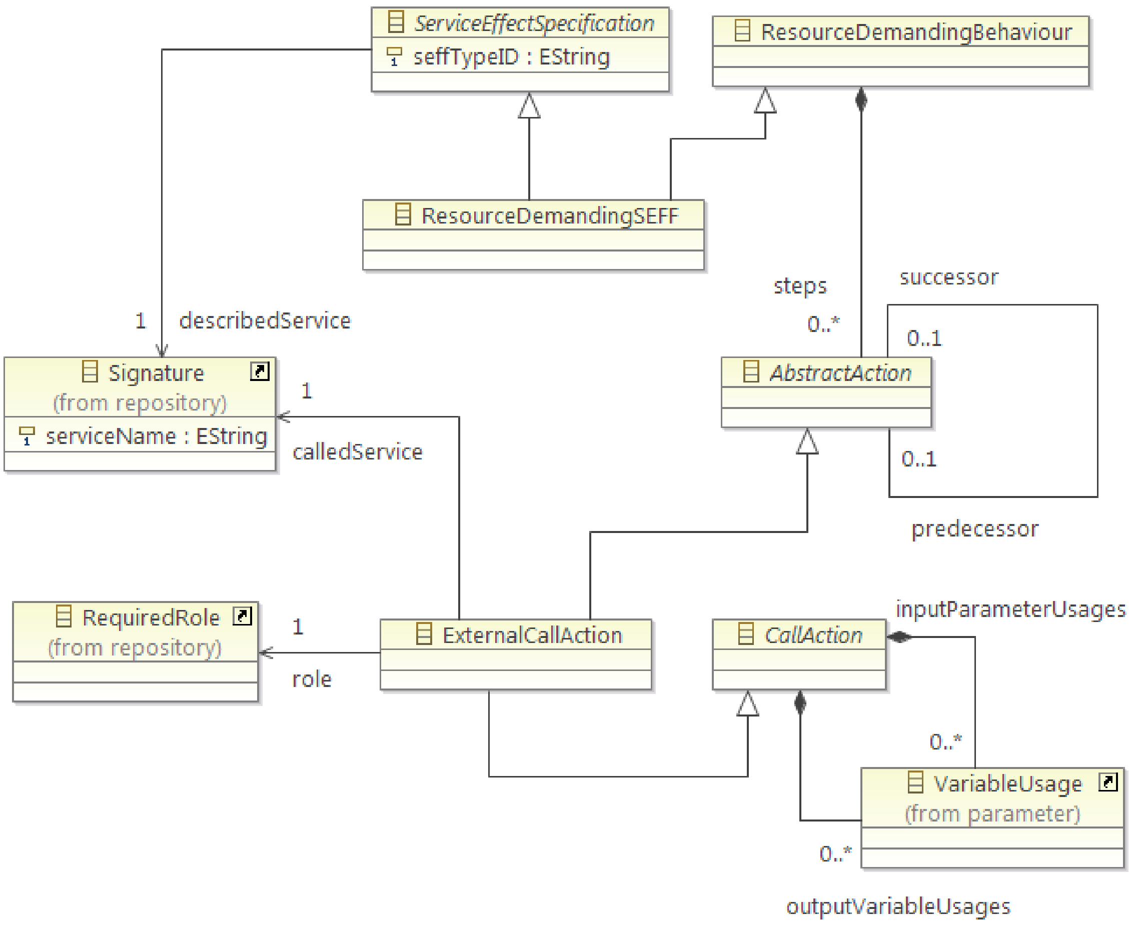


Figure 5.10: ExternalCallAction PCM

6. If *A.providingAssemblyContext.encapsulatedComponent* is of type *ComposedStructure*, follow the steps of the previous algorithm used for the *EntryLevelSystemCall* with *A.providedRole* as *P*.

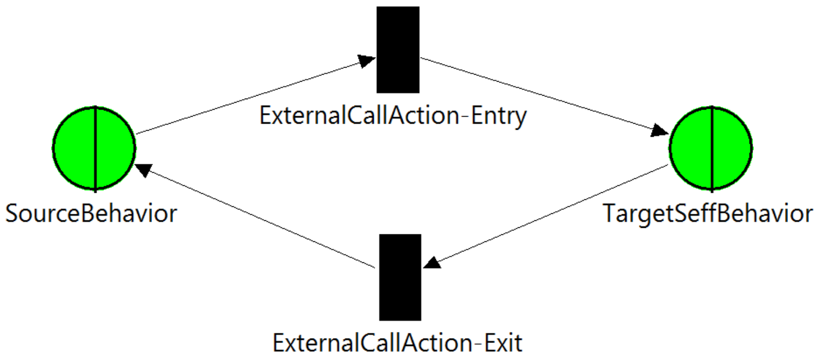


Figure 5.11: ExternalCallAction QPN

The QPN result of the ExternalCallAction mapping is almost identical to the EntryLevelSystemCall mapping. As we can see in Figure 5.11, the only notable differences are that the transitions are now called *ExternalCallAction-Entry* and *ExternalCallAction-Exit* and that we find the subnet of a *ResourceDemandingBehavior* at the source side of the mapping. Again, a new color is generated and used for the *TargetSeffBehavior* part of the mapping to distinguish it from other calls. Likewise the *VariableUsage* can affect the resource demands at *InternalActions*.

Apart from general limitations that apply to the use of stochastic variables (Section 5.1) no other limitations are known for this part of the mapping.

5.2.4 Branches

A branch routes an incoming request to exactly one of its child behaviors. The behavior is not deterministic and depends on the probabilities of the different child behaviors. The probabilities must add up to 1.0. There are three different kinds of branches in the PCM meta-model. *Branch* from the usage model and to configurations of *BranchAction* inside SEFFs. A *BranchAction* uses either only *ProbabilisticBranchTransitions* or only *GuardedBranchTransitions*. All three PCM variants are mapped to a single QPN variant.

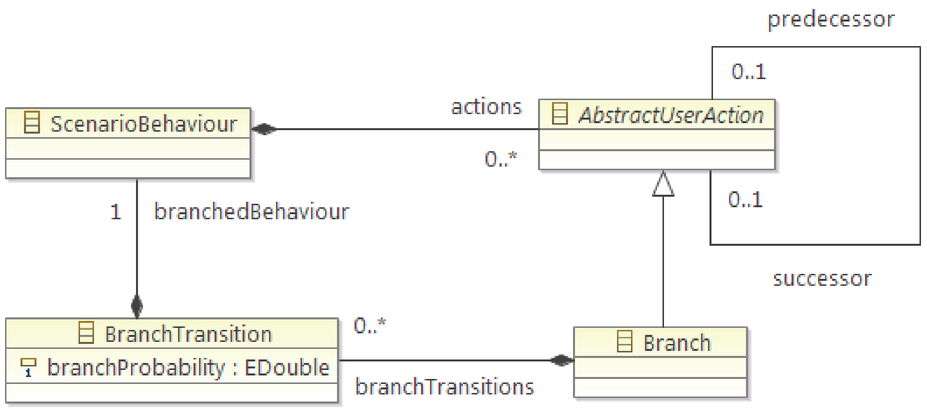


Figure 5.12: Usage Model Branch PCM

Figure 5.12 shows the *Branch* entity from the usage model. It contains a number of *BranchTransitions*, each having a probability *branchProbability* of type double. Each *BranchTransition* also contains a child *ScenarioBehavior*.

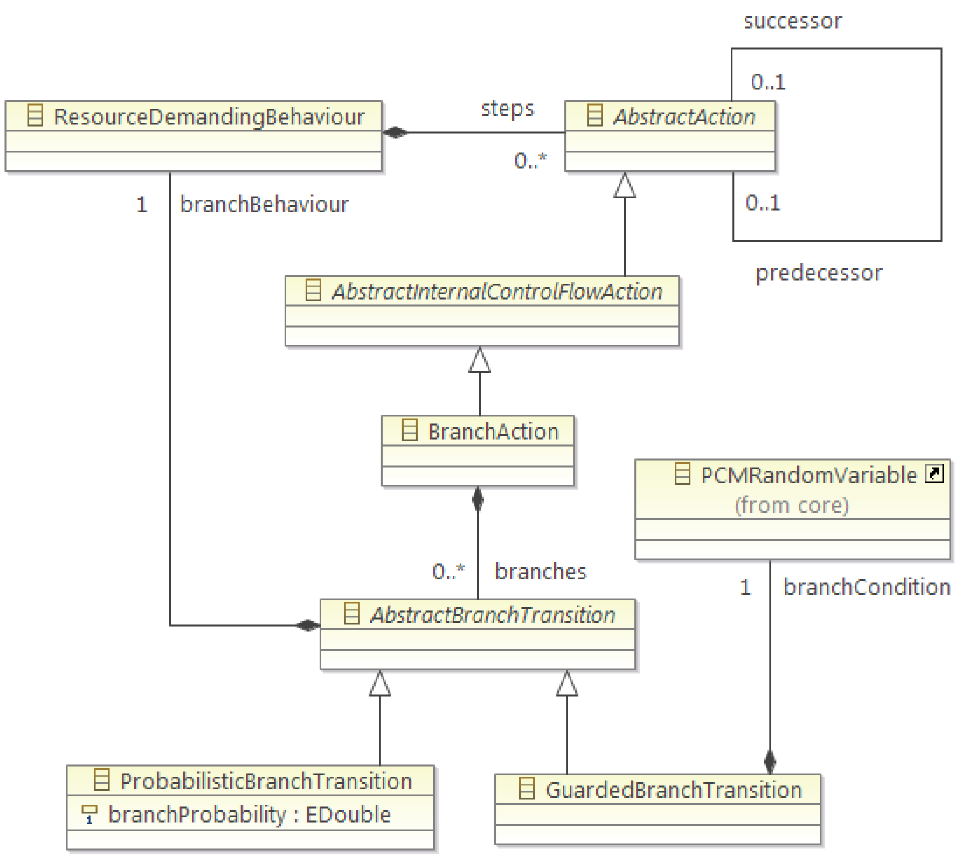


Figure 5.13: BranchAction PCM

The situation is only slightly different for *BranchAction*. As we can see in Figure 5.13, a *BranchAction* with *ProbabilisticBranchTransitions* corresponds exactly to the usage model *Branch*. Only the class of the behavior (*ResourceDemandingBehavior* instead of *ScenarioBehavior*) and the super class (*AbstractAction* instead of *AbstractUserAction*) differ. When *GuardedBranchTransitions* are used the probabilities are not specified directly. Instead, *branchCondition* is a stochastic variable specification that must evaluate to true or false. As the individual tokens in a QPN have no other attributes apart from their color, the condition for each request cannot be evaluated at simulation time. The DependencySolver evaluates the condition in a preprocessing step (Section 5.1) and computes the probabilities of each branch for each request class.

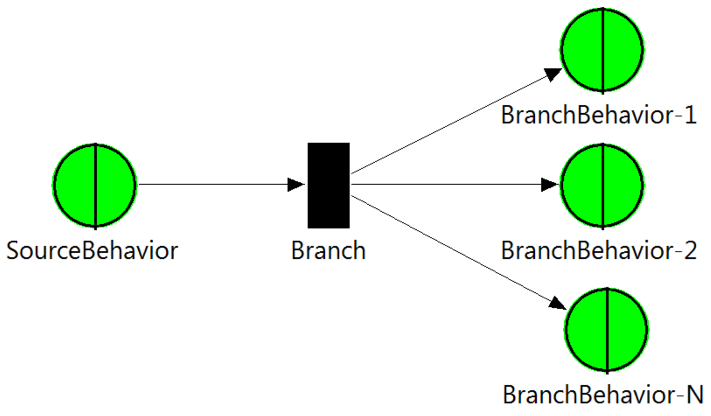


Figure 5.14: Branch QPN

All three branch types can now be reduced at simulation time to a set of N child behaviors B_i with corresponding branching probabilities P_i. Figure 5.14 shows the QPN result in a generalized fashion. From the last place of the subnet of the *predecessor*, the *Branch* transition consumes a token. The transition has N modes corresponding to the number of child behaviors. Each mode creates a token in the first place of the subnet for child behavior B_i and has a firing weight of the branching probability P_i. Not shown in the figure are the transitions that lead back to the successor of our branch entity. The correct place is passed as a parameter to the mappings of the child behaviors and they use the given place as the exit place of the mapping.

A limitation applies to nested *BranchAction*s with *GuardedBranchTransition*s. This means that a guarded branch *Bchild* is used in the child behavior of another guarded branch *Bparent*. If a single stochastic variable is used in the *branchCondition*s of both *Bchild* and *Bparent*, this can result in inaccurate behavior. The problem is that the DependencySolver does not take into account the statistical dependency which results from evaluating a condition which depends on the outcome of another condition. For example, if *Bparent* contains a *GuardedBranchTransition* with the guarding specification 'a.BYTESIZE < 100'. We assume that 'a.BYTESIZE' is a stochastic expression variable available in the current context. Let us also assume that the condition holds in 50% of the cases. If *Bchild* now uses the condition 'a.BYTESIZE >= 100', the dependent probability would be 0. If 'a.BYTESIZE' would be larger than 100, we would have never entered that branch. The DependencySolver does not take this into account and the probability of this branch transition is again set to 50%.

5.2.5 Loops

A loop in the PCM meta-model generally has the following properties:

1. The loop has a single child behavior.
2. The number of times the loop child behavior is executed is specified by a stochastic expression that evaluates to an integer constant I or to an integer type probability mass function (IntPMF). An IntPMF has N possible integer values V_i that each have a probability P_i. All probabilities P_i must add up to 1.0. In the following, the constant case is treated as $N = 1$, $V_i = I$ and $P_i = 1.0$.
3. The next loop iteration does not start until the previous request has completed the child behavior.

With the help of the DependencySolver (Section 5.1) all different PCM loop entities (*Loop* from the usage model, *LoopAction* and *CollectionIteratorAction* in SEFFs) are reduced to the presented information. Therefore after discussing the PCM entities, the QPN result is presented in a generalized fashion. The actual mapping only adapts the place and transition names to match the host entity.

Figure 5.15 shows the usage model *Loop* entity. It directly contains a *ScenarioBehavior* and a *PCMRandomVariable* holding the loop iteration specification.

The *LoopAction* contains a *ResourceDemandingBehavior* and a *PCMRandomVariable* as the iteration count specification accordingly (Figure 5.16). The *bodyBehavior* property is shared with the *CollectionIteratorAction* and is therefore located in the super class *AbstractLoopAction*.

Figure 5.17 shows that a *CollectionIteratorAction* does not have a direct loop iteration specification. Instead, a *Parameter* is referenced that belongs to the *Signature* that the

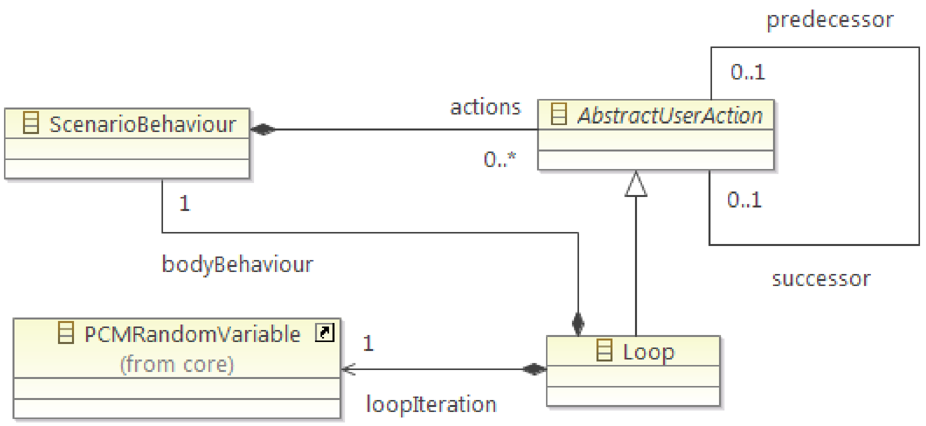


Figure 5.15: Usage Model Loop PCM

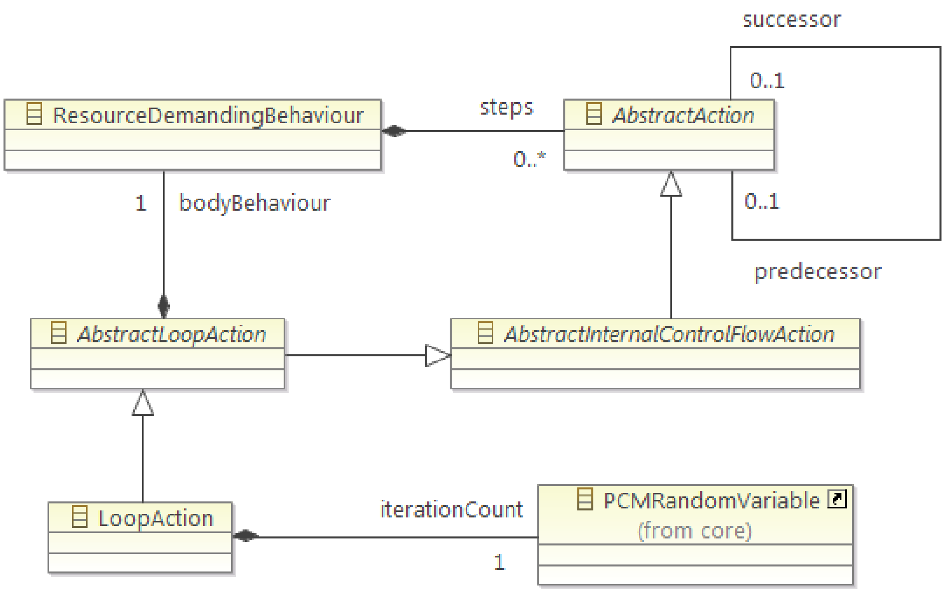


Figure 5.16: LoopAction PCM

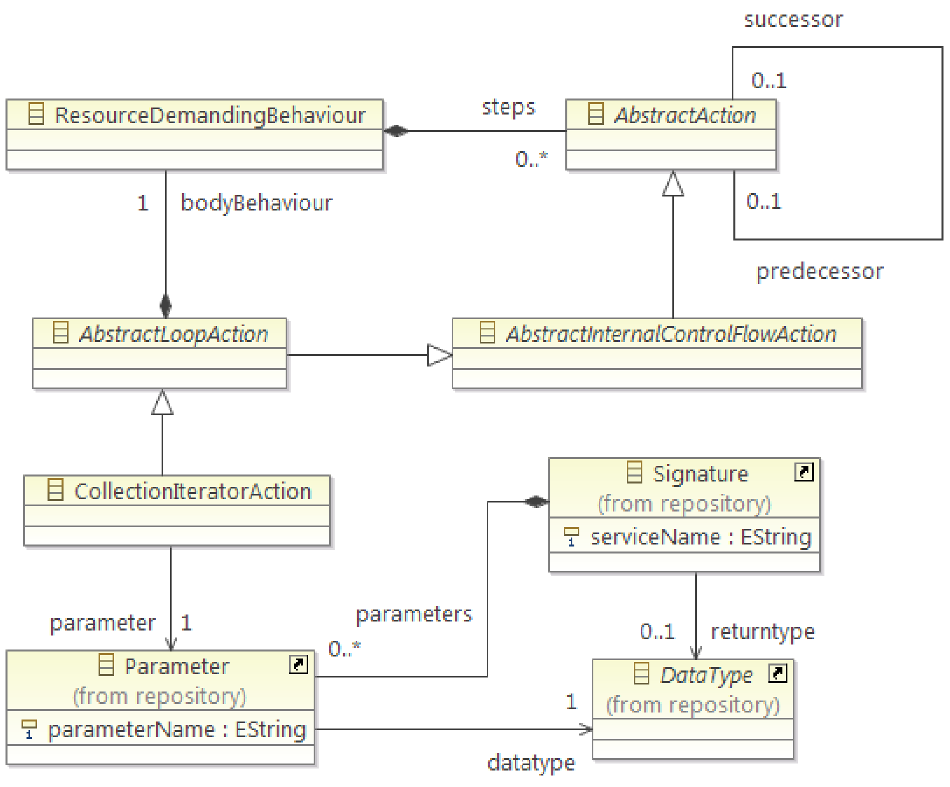


Figure 5.17: CollectionIteratorAction PCM

surrounding SEFF implements. The *datatype* of the *parameter* must also be of type *CollectionDataType*. The loop iteration specification is then implicitly derived from the expression 'parameterName.NUMBER_OF_ELEMENTS' where 'paramterName' is the value of the *parameterName* property of *parameter*.

For the mapping of loop structures to QPNs, two different possibilities have been considered. Figure 5.18 shows the first solution, which has been implemented in the mapping transformation.

Figure 5.19 shows the alternative mapping. As the two alternatives share considerable parts, those parts will be discussed first. Then the differences, benefits and limitations of each approach are discussed.

Both mappings are not trivial as in a QPN we have only local information about tokens. Any request property must be encoded in the token color. Local information means that a token has an identity that is local to a place. This includes information about the order of tokens.

Both loop subnets can be divided into an inner and an outer part. The outer part consists of the *Loop-Entry* and *Loop-Exit* transitions, as well as of the *Loop-Pool* and *Loop-Depository* places. The inner part consists of the *Loop-Inner-Entry* and *Loop-Inner-Exit* transitions, as well as of the *Loop-Inner-ColorCode* place. The outer part handles the token input from the *predecessor* and token output to the *successor*. The inner part handles the input and output to and from the child behavior, denoted *LoopBehavior*. The color of the input and ouput tokens is not determined in this part of the mapping and will be referred to as the loop input color.

The outer and inner part separation is necessary to implement property number 2. If only one token color was used, the exit transition would not know when the whole request is

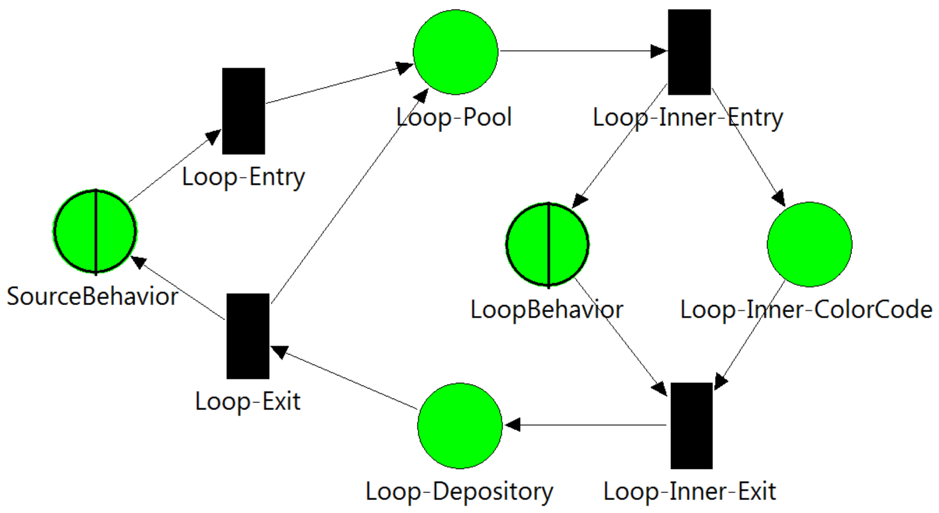


Figure 5.18: Loop QPN

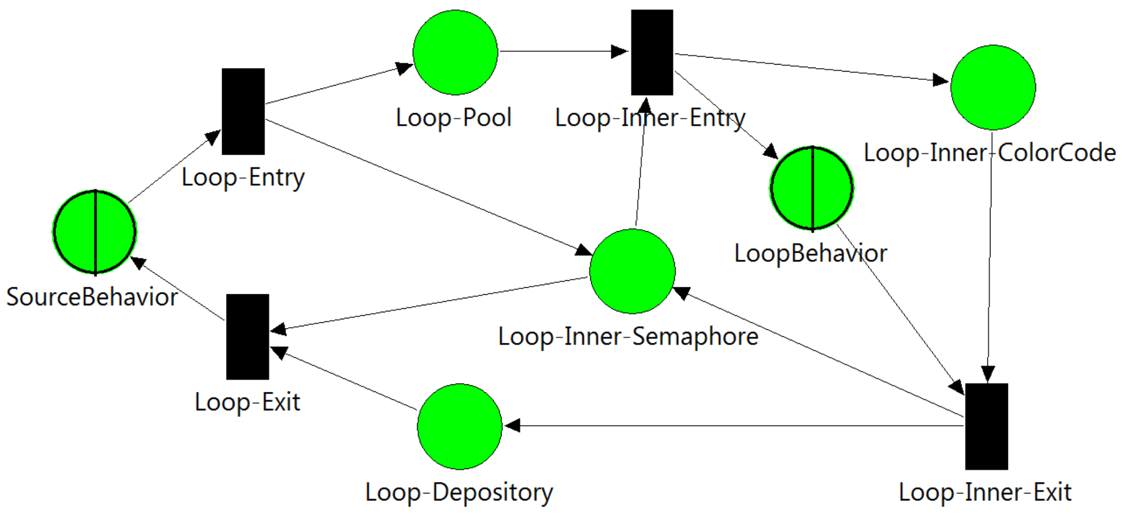


Figure 5.19: Loop QPN (Alternative)

complete. Different iteration counts are possible. This is decided at the loop entry and encoded into a new color. For each of the N possibilities of iteration counts one color C_i and one mode M_i is generated in the *Loop-Entry* transition. The firing weight of each mode is set to P_i. The *Loop-Inner-Entry* transition takes a token of color C_i from the *Loop-Pool*, generates a token of the loop input color in the first place of the loop child behavior denoted as *LoopBehavior*, and generates a token of color C_i in *Loop-Inner-ColorCode*. The loop input color is used in the *LoopBehavior* to limit the number of modes and colors in the child behavior subnet to a minimum. The iteration count information encoded in C_i is needed only locally in this part of the mapping, not inside the subnet representing the child behavior. The color code place is then necessary for the *Loop-Inner-Exit* transition to know which color C_i to generate in the *Loop-Depository* when consuming a token of loop input color from the last place of the *LoopBehavior*.

The other parts of the mapping differ in how property number 3 is implemented. The first approach uses two different modes per color C_i in the *Loop-Exit* transition. One mode to leave the loop and one mode to return the token of color C_i to the *Loop-Pool* for another client behavior iteration. The exit mode has a firing weight of P_n where n is the iteration count of color C_i. The return mode has a firing weight of $1 - P_n$. The random selection between the two modes for color C_i at the *Loop-Exit* transition behaves like a Bernoulli random variable. The number of iterations until the loop is left are therefore geometrically distributed with an expected value of $1/P_n$. Therefore we choose $P_n = 1/n$. The mean number of times that a request completes the inner behavior now equals the expected value $1/P_n = 1/(1/n) = n$. The limitation is that for an individual request the number of times the internal behavior is executed does not necessarily equal n.

The alternative approach does not make this simplification. The number of times the loop behavior is executed always exactly equals the target n for each request. This is achieved by generating not one, but n tokens of color C_i in the *Loop-Pool*. The maximum number of parallel executions of the loop body equals the number of requests in the loop subnet. Therefore an additional semaphore place, *Loop-Inner-Semaphore*, with color references for all colors C_i and an initial population of 0 for each color is introduced.

Loop-Inner-Entry cannot place tokens freely in the *LoopBehavior*, it needs one token from the semaphore of the color that matches the consumed token. *Loop-Inner-Exit* puts a token of matching color back after a token finishes the loop behavior. The maximum number of tokens in the semaphore place and thus inside the loop behavior should always equal the number of PCM requests in the loop structure. Therefore, *Loop-Entry* generates in the semaphore place one token of the same color C_i which it generates inside the *Loop-Pool*. *Loop-Exit* takes one token of color C_i away from the semaphore place when it has consumed n tokens of matching color C_i from the *Loop-Depository* to complete the request.

The use of color C_i inside the semaphore place ensures that the loop body is executed with a probability that is independent of the iteration count. Only tokens that have a token of matching color in the semaphore can enter the child behavior. If the tokens in the *Loop-Pool* had an equal probability of entering the child behavior, this would statistically favor requests with higher iteration counts.

The first alternative (generating only one token for each color) was chosen for the implementation. It has the benefit of using less places, modes and tokens.

A limitation applies to the mapping of *CollectionIteratorAction*. It carries a special semantic which is not implemented. The extra semantic compared to a *LoopAction* is that stochastic variables that are used in the loop body must be evaluated in a statistically dependent manner. For example, let ‘a.BYTESIZE’ contain an IntPMF that has two possible values a and b. If anywhere in the loop behavior ‘a.BYTESIZE’ is used, it is evaluated and if a is returned, all other references to 'a.BYTESIZE' must then also return a. This special semantic is not mapped as this is not possible for continuous variables.

To map the semantic, the modes in the *Loop-Entry* transitions could not simply have the firing weights according to the probabilities. If one loop randomly choses $i = 1$ for a reference to a stochastic variable i, all other loops in the behavior also have to chose $i = 1$. A central place would have to be introduced that is filled with the right color and is used by all accessors. This would have to be implemented for all stochastic variables and their combinations that result from the different stochastic expressions. An expression 'a.BYTESIZE * b.BYTEZISE' would have four different outcomes if both 'a.BYTESIZE' and 'b.BYTESIZE' had two possibilities each. In a continuous case the behavior simply cannot be implemented as no continuous properties are available for tokens. The token color is discrete and has to be decided upon in advance. Computations with token colors are not possible.

SimQPN contains an experimental feature called 'probes', which allows to track individual tokens (see Section 2.2.5). When probes are used to measure the response time distribution instead of just the mean response time, both alternatives have limitations. In the first variant, the number of loop iterations is not guaranteed for individual requests. This causes a spread in the measured mean response times. The second variant, though guaranteeing the right loop iteration count, has a different problem that also causes the request response times to spread. The problem lies in the fact that *Loop-Exit* consumes n tokens of matching color C_i from the *Loop-Depository.* The probes feature adds a timestamp and a probe id to affected tokens. The timestamp is selected at random from all incoming tokens. This causes a spread in response time because tokens generated for one requests can now be used for completing a different request. A similar situation arises in *Loop-Inner-Exit*, as it consumes tokens from both the loop child behavior and from *Loop-Inner-Color-Code*. During prototypical tests, the error introduced by the *Loop-Inner-Exit* merge was insignificant compared to the other errors.

5.2.6 Forks

A fork represents behavior executed in parallel. An incoming request is split into a number of child requests started simultaneously. If the original request completes immediately, having only started the child requests, we speak of an asynchronous fork. If the original request has to wait until the child requests complete, we speak of a synchronous fork. Both types of forks are supported by PCM but they are realized by a single entity, the *ForkAction.*

Figure 5.20 shows the *ForkAction* and related entities. A *ForkAction* can contain directly a number of N *ForkedBehavior*s. In addition, it can contain a *SynchronizationPoint* which then contains M more *ForkedBehavior*s. The behaviors contained directly are executed asynchronously in contrast to the behaviors contained in the *SynchronizationPoint*, which are executed synchronously. The *SynchronizationPoint* has an additional reference *outputParameterUsage* which is handled exclusively by the DependencySolver (Section 5.1). It allows the different synchronized control flows to return a parameter which is stored in the context of the original request. This is not possible for asynchronous control flows as the request is returned immediately.

In Figure 5.21 we see the QPN representation of a *ForkAction.* There are three transitions. *ForkAction-Split* consumes a token from the last place of the *predecessor* subnet and creates a token in the start places of each of the $M + N$ child behaviors. *ForkAction-Consume-Asynchronous* consumes tokens from the end places of the N asynchronous child behaviors, but does not create any new tokens. *ForkAction-Join-Synchronous* waits until a token is

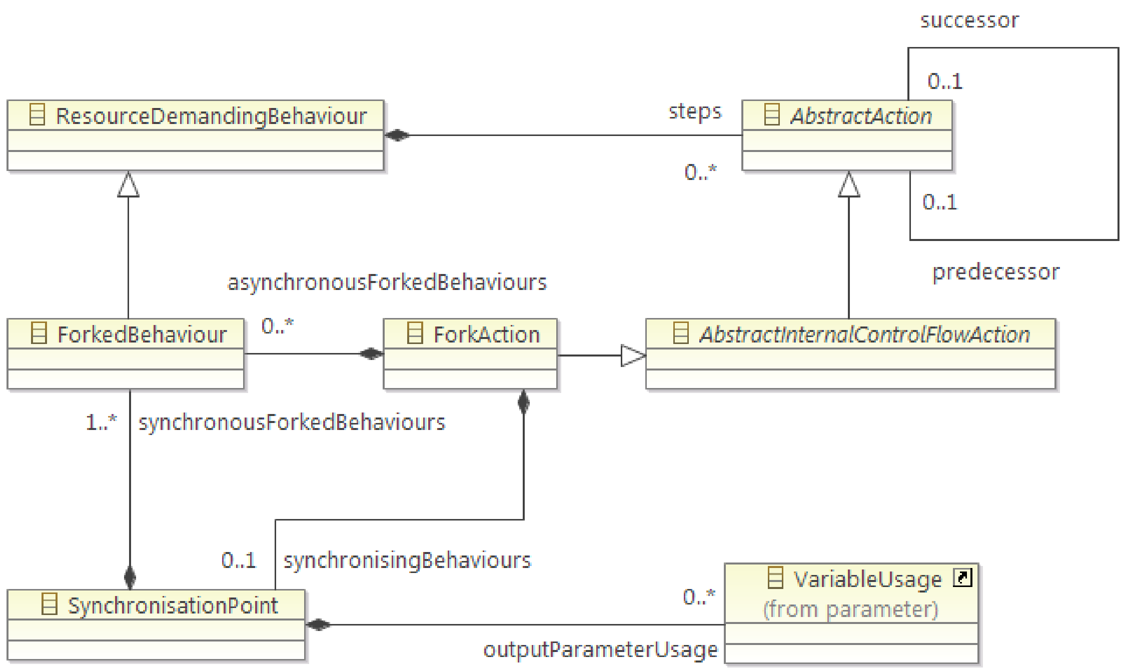


Figure 5.20: ForkAction PCM

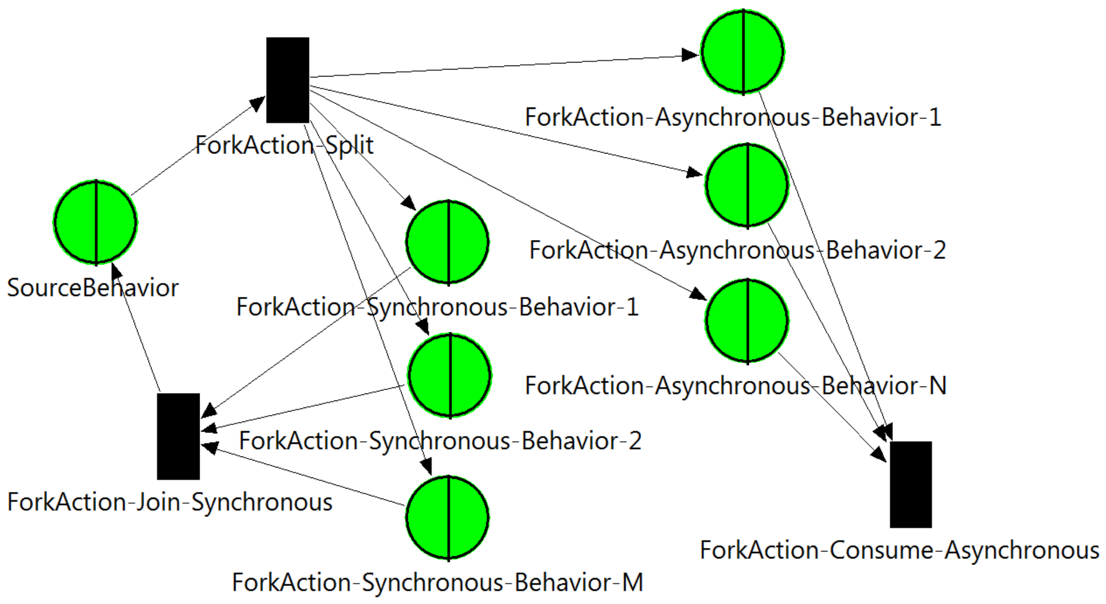

ForkAction-Split
ForkAction-Asynchronous-Behavior-1
ForkAction-Asynchronous-Behavior-2
ForkAction-Asynchronous-Behavior-N
SourceBehavior
ForkAction-Synchronous-Behavior-1
ForkAction-Synchronous-Behavior-2
ForkAction-Synchronous-Behavior-M
ForkAction-Join-Synchronous
ForkAction-Consume-Asynchronous

Figure 5.21: ForkAction QPN

available in each of the M end places of the synchronous child behaviors. It then consumes all of them and creates a new token in the start place of the *successor* subnet.

The case in which the *ForkAction* does not have a *SynchronizationPoint* ($M = 0$) is handled by adding a dummy ordinary place in place of a synchronized client behavior. Without the dummy place the token representing the original request, consumed when starting the asynchronous requests, would be lost.

A limitation applies to the mapping of synchronized forked behavior. The synchronization of two sub-requests generated by a single host request cannot be exactly mapped to QPNs. Individual tokens carry no identity and it cannot be decided for two tokens whether or not they belong to the same host request. The same problem is encountered when tracking individual tokens using the probes feature (Section 2.2.5). Likewise, the tokens are consumed without considering their host request, introducing an error. The extent of the error depends both on the number of parallel behaviors and on the properties of the child behavior subnets.

5.2.7 Processing Resources

Processing resources normally represent physical resources with a certain processing rate and a certain scheduling strategy for requests. Each request has a distinct resource demand (in seconds), the time the request occupies the resource until completion. When one request is being served, other requests have to wait in a queue, except in the case of a delay resource, in which case each request is immediately served and there is no contention. A CPU or a hard drive are good examples of processing resources.

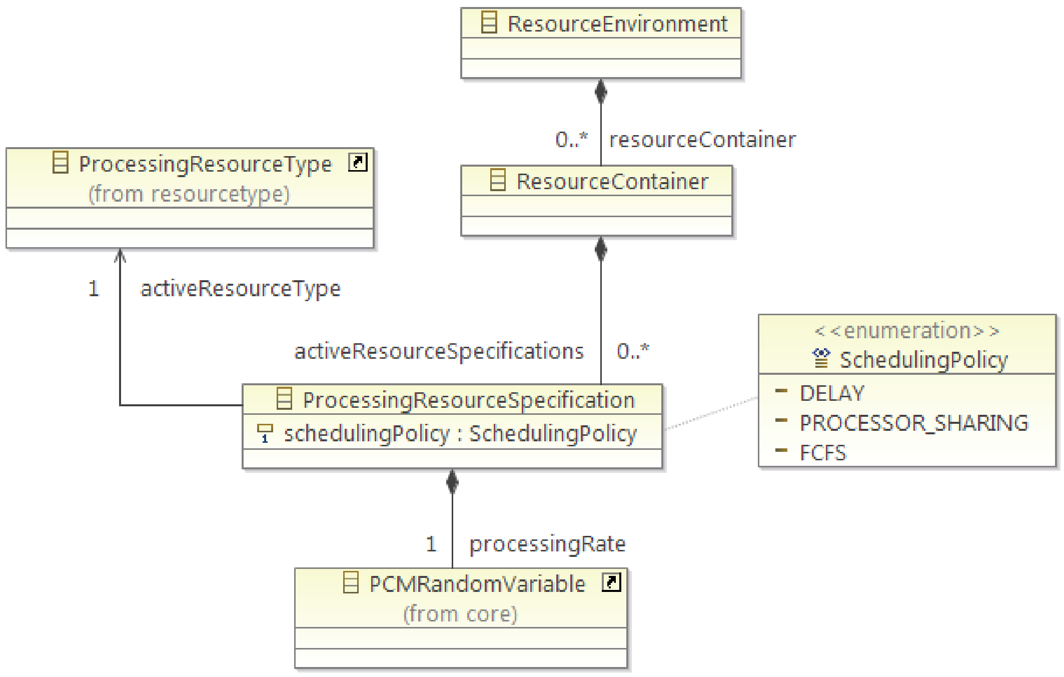


Figure 5.22: ProcessingResourceSpecification PCM

The PCM version employed for this thesis supports only single-server processing resources and three scheduling strategies: first-come-first-served (FCFS), processor-sharing (PROCESSOR_SHARING) and delay (DELAY). As we can see in Figure 5.22, each *ResourceContainer* of the *ResourceEnvironment* can have an arbitrary number of *ProcessingResourceSpecifications* which each have a *ProcessingResourceType* and a *processingRate*. It is not possible to have two *ProcessingResourceSpecifications* in the same *ResourceContainer* that reference the same *ProcessingResourceType*. In that context the *names* of the referenced *ProcessingResourceTypes* must also be unique. This is necessary because a *ProcessingResourceSpecification* is referenced indirectly using the current *AssemblyContext*, *AllocationContext* and *ProcessingResourceType*.

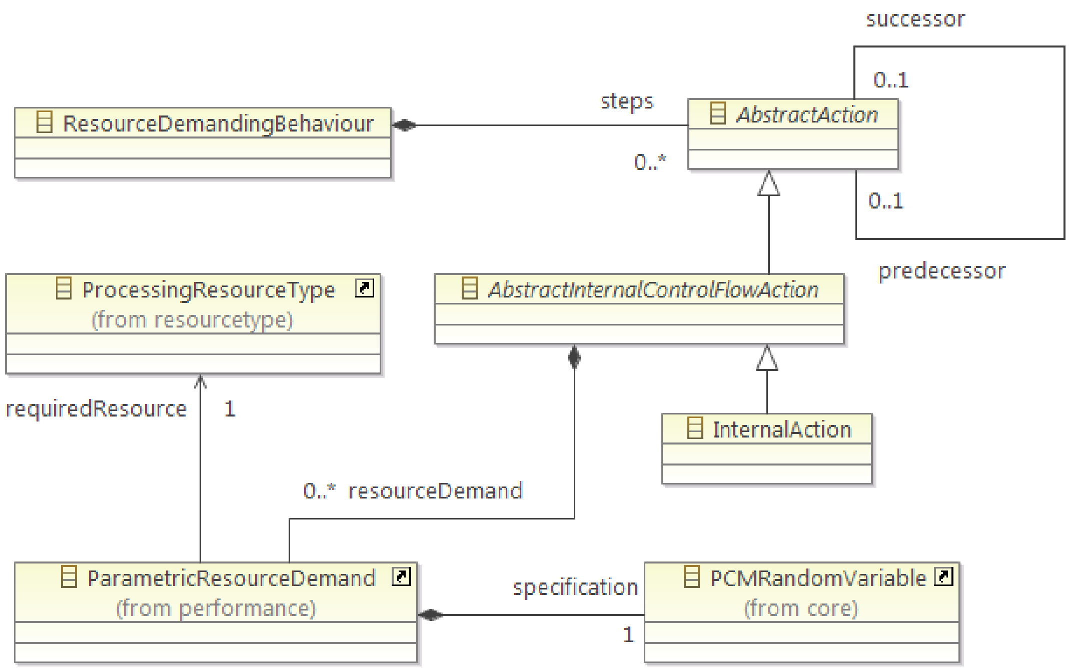


Figure 5.23: InternalAction PCM

Figure 5.23 shows *InternalAction*, an entity that represents a load that a request places on one or more processing resources. It contains a number of *N ResourceDemands*. Each *ResourceDemand* contains a resource demand *specification* in the form of a *PCMRandomVariable* and a reference to a *ProcessingResourceType*. Contrary to how a resource demand was introduced (time in seconds), the *specification* is in abstract units and must logically match the processing rate of the target *ProcessingResource*. The actual resource demand times are determined at simulation time.

The figure shows that the *resourceDemands* property actually belongs to the abstract base entity *AbstractInternalControlFlowAction* which is also the base entity of several other *AbstractActions* we have discussed. The PCM-Bench editors, however, allow adding a *ResourceDemand* to an *InternalAction* only and therefore other possibilities were not considered for the mapping.

As mentioned before, an *InternalAction* does not reference a *ProcessingResourceSpecification* directly. The component behavior, the system assembly and the system allocation information are separated in PCM. An *InternalAction* is part of a SEFF which describes the behavior depending on the stochastic variables passed to it. The SEFF belongs to a *BasicComponent* which is referenced by an arbitrary number of *AssemblyContexts*. The *AssemblyContext* is itself part of a bigger composition hierarchy, which is described in detail in Section 5.2.3. We assume that during the mapping the assembly context information is available and we know the correct *AssemblyContext*.

Figure 5.24 shows how the allocation context information is organized. Each *AllocationContext* maps one *AssemblyContext* to one *ResourceContainer* of the *ResourceEnvironment*. With this information the right *ProcessingResourceSpecification* is uniquely determined at mapping time.

Figure 5.25 shows the QPN subnet which is generated for an *InternalAction*. The transi-

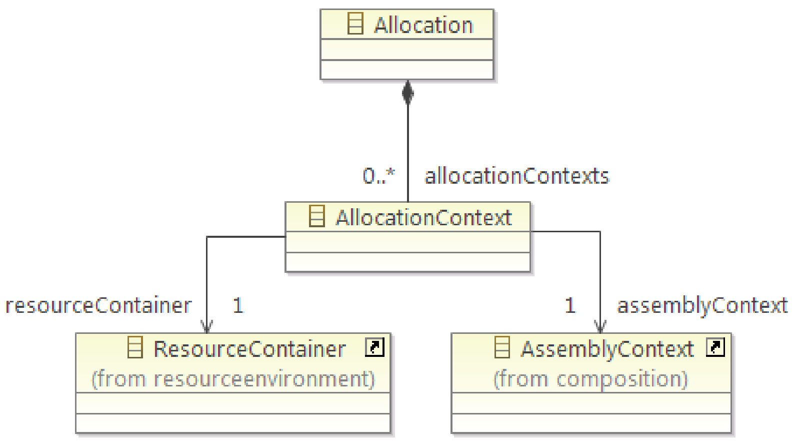


Figure 5.24: Allocation PCM

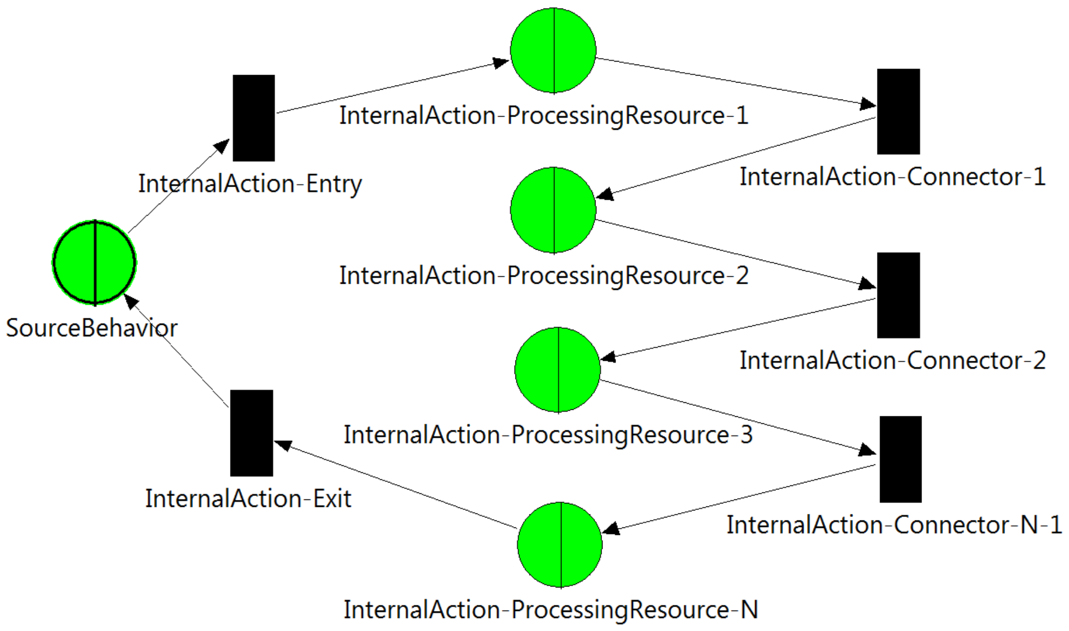

InternalAction-ProcessingResource-1
InternalAction-Entry
InternalAction-Connector-1
InternalAction-ProcessingResource-2
SourceBehavior
InternalAction-Connector-2
InternalAction-ProcessingResource-3
InternalAction-Exit
InternalAction-Connector-N-1
InternalAction-ProcessingResource-N

Figure 5.25: InternalAction QPN

tions *InternalAction-Entry* and *InternalAction-Exit* handle token input from the *predecessor* and token output to the *successor* respectively. The resource demands are processed in series, one after another. The order cannot be specified. This matches the behavior of the SimuCom simulator. For each of the N *ResourceDemands* R_i of the *InternalAction* a queueing place *InternalAction-ProcessingResource-i* is generated. For each R_i with $i < N$ a connector transition *InternalAction-Connector-i* is also generated. The DependencySolver (Section 5.1) has solved all parametric dependencies in the *specification* of R_i and provides an empirical distribution. The distribution is then divided by the *processingRate* of the target *ProcessingResourceSpecification* defined by the current context (using the folding module of the DependencySolver). This results in the resource demand distribution for the tokens of the current color. The distribution is used for an empirical distribution in the color reference of the queueing place for R_i.

The target queues of the N queueing places are only generated on demand, one queue for each *ProcessingResourceSpecification* of each *ResourceContainer*. The scheduling disciplines are mapped to their respective QPN queue scheduling strategies. The number of servers is set to 1.

If $N = 0$, a dummy ordinary place is generated in place of the queueing places in order not to lose the request.

Formally there are no limitations for this part of the mapping. However, as for the workload inter-arrival times and think times (discussed in Section 5.2.2), mean resource demands and a deterministic distribution are used inside the corresponding queueing places instead of the empirical distribution. This is only a temporary implementation limitation that will be fixed once an improved version of SimQPN is available.

5.2.8 Passive Resources

Passive resources normally represent software resources of which there is only a limited number available. A request might need one of these resources for a certain part of its execution. If more request reach that section than instances are available, requests have to wait until the next instance becomes available. Good examples of passive resources are a thread pool or a pool of database connections.

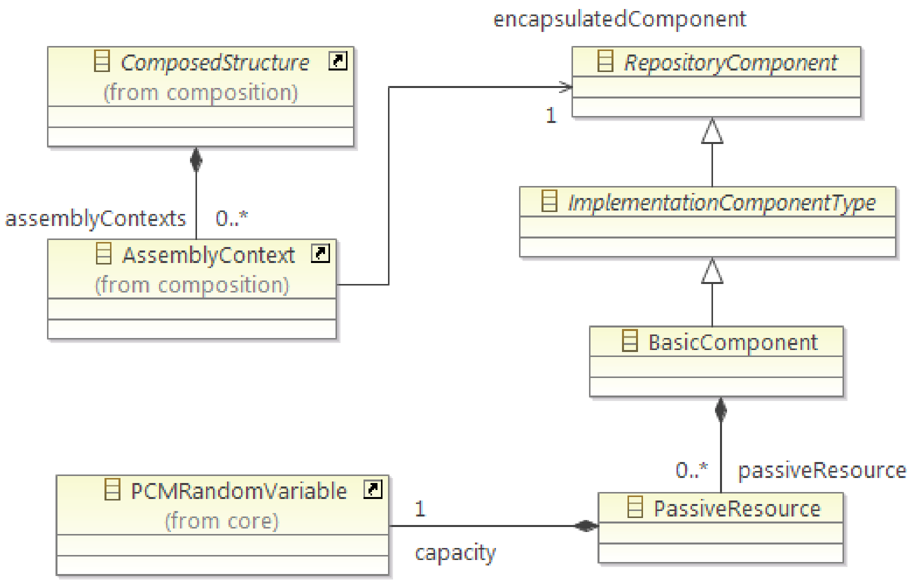


Figure 5.26: PassiveResource PCM

Figure 5.26 shows the PCM representation of passive resources. A *BasicComponent* contains a number N of *PassiveResources*. Each *PassiveResource* contains an initial *capacity* specification of type integer. While *PassiveResources* are defined per component, they are instantiated per *AssemblyContext*.

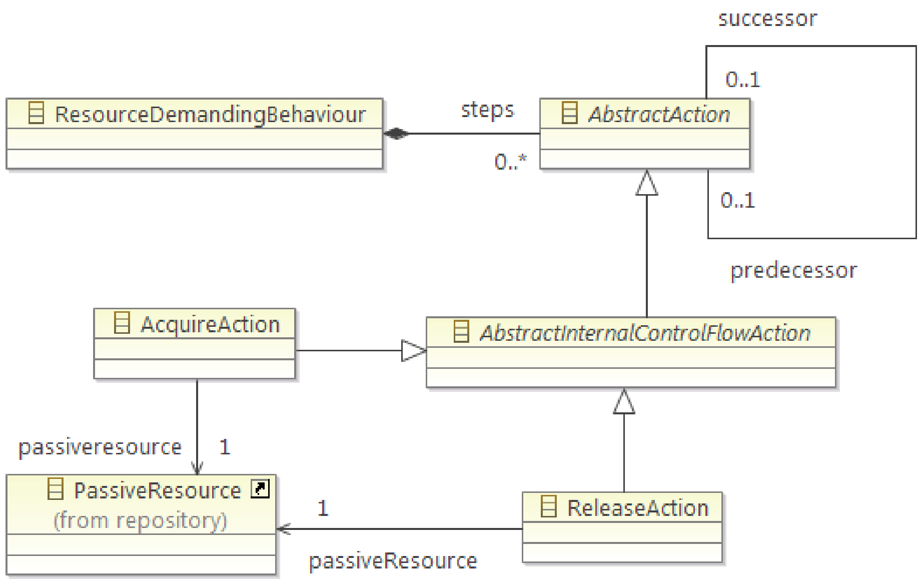


Figure 5.27: AcquireAction and ReleaseAction PCM

The two entities that mark the section during which a request requires a passive resource are *AcquireAction* and *ReleaseAction*, depicted in Figure 5.27. They both reference one of the N *PassiveResources*. *AcquireAction* marks the beginning of the section. The current capacity is reduced by one. When the current capacity reaches 0, the request has to wait. *ReleaseAction* marks the end of the section, increasing the current capacity of the *PassiveResource* again.

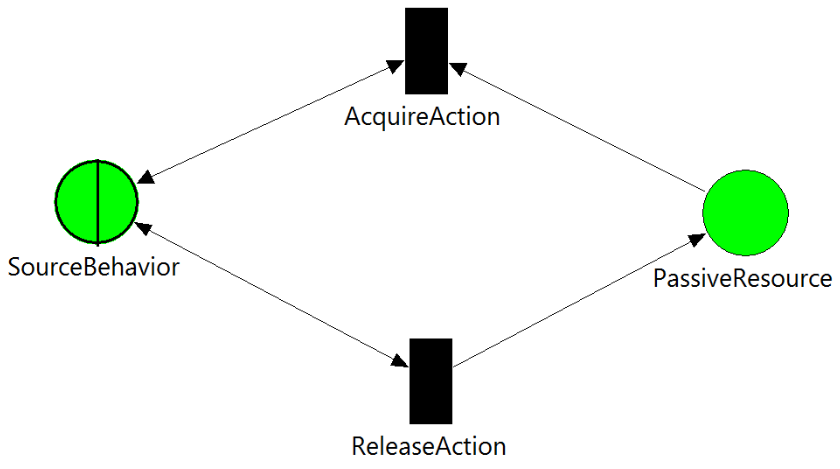


Figure 5.28: AcquireAction and ReleaseAction QPN

Figure 5.28 shows the resulting QPN subnet for both *AcquireActions* and *ReleaseActions*. An *AcquireAction* is represented by the *AcquireAction* transition. It connects the last place of the *predecessor* with the first place of the *successor* and the ordinary place *PassiveResource* representing the *PassiveResource*. The initial number of tokens of the *PassiveResource* place is set to the *capacity*. The *AcquireAction* consumes a token from the *predecessor* and one token from the *PassiveResource* place and generates one token in the *successor*. Similarly, the *ReleaseAction* transition connects its *predecessor* and *successor*, generating a token in the *PassiveResource* place for each token it consumes from the *predecessor*.

As mentioned before, the actual *PassiveResource* place that is used for the mapping depends on the current *AssemblyContext*. If three *AssemblyContexts* reference a *BasicComponent* containing a *PassiveResource*, there will be three instances of the *PassiveResource* place. For simplicity, this is not shown in the figure.

Only general limitations regarding the stochastic expression of the *PassiveResource capacity* apply. *AcquireAction* and *ReleaseAction* can be used anywhere in a a *ResourceDemandingBehavior*. It is assumed, that the user sets them up in a meaningful way (and without deadlocks).

5.2.9 Linking Resources

A linking resource represents any kind of network between two or more resource containers. In the resource model chosen for PCM it has a processing rate and a latency. When a request is made across container boundaries, first the combined bytesize of all input parameters is placed on the linking resource. After the request is completed at the target resource, the combined bytesize of all output parameters (including the return parameter) needs to be processed by the linking resource. In addition, a delay is added to the request. In PCM a linking resource is represented by a *LinkingResource* entity. Calls between containers are represented by *ExternalCallActions* (Section 5.2.3) that target a component on a different *ResourceContainer*.

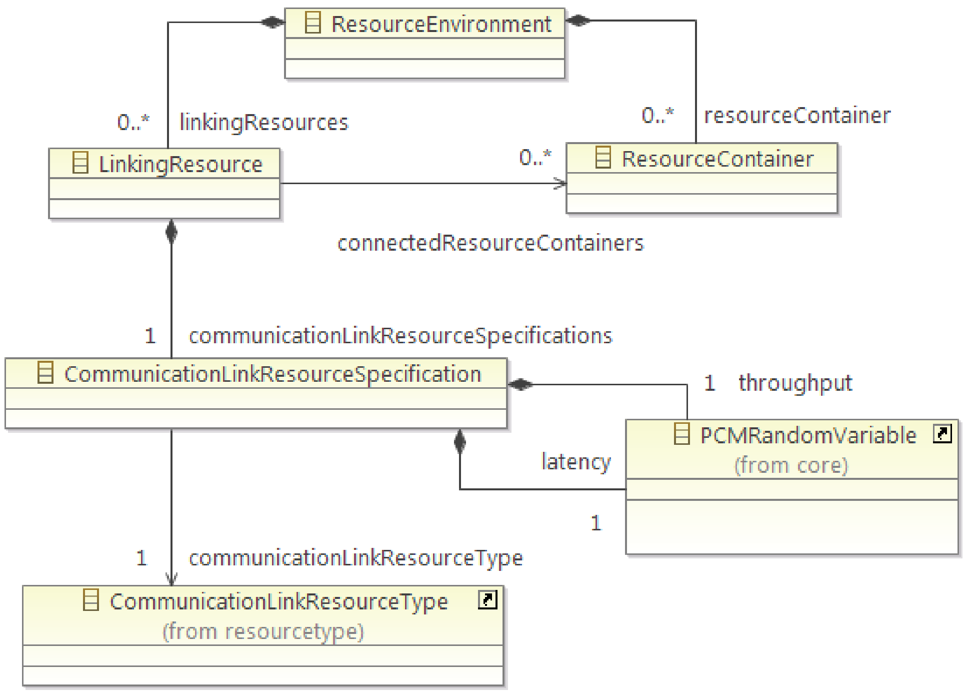


Figure 5.29: LinkingResource PCM

Unlike a *ProcessingResourceSpecification* (Section 5.2.7), a *LinkingResource* is contained directly by the *ResourceEnvironment*. It is on the same level as a *ResourceContainer*. This becomes obvious when we consider that a *LinkingResource* connects a number of *ResourceContainers* (the *connectedResourceContainers* property). As we can see in Figure 5.29, a *LinkingResource* also contains a *CommunicationLinkResourceSpecification* which acts as a container for the other properties of the *LinkingResource*. It has a *communicationLinkResourceType* and it has *throughput* and *latency* specifications in the form of *PCMRandomVariable*s. The *throughput* is specified in bytes per second. The latency is specified in seconds.

As the linking resource is only used for inter-component requests, which happen only at *ExternalCallAction*s, the mapping for *ExternalCallAction*s is extended by a number of *LinkingResource* dependent parts. At an *ExternalCallAction* we first have to identify the target resource container. It is assumed that the current *AssemblyContext* is available and that the assembly context of the target SEFF is known. More details on the assembly composition can be found in Section 5.2.3. For both *AssemblyContext*s, the *AllocationContext* is looked up (see Figure 5.24) which then links to the *ResourceContainer*s of both the source and the target context. If the containers match, no linking resource mapping is executed. If they do not match, one *LinkingResource* is selected from all of the *LinkingResource*s of the *ResourceEnvironment* which includes both the source and the target *ResourceContainer* in its *connectedResourceContainer*s set. It is assumed that no two *LinkingResource*s contain the same pair of *ResourceContainer*s. If a *LinkingResource* is found, the mapping proceeds.

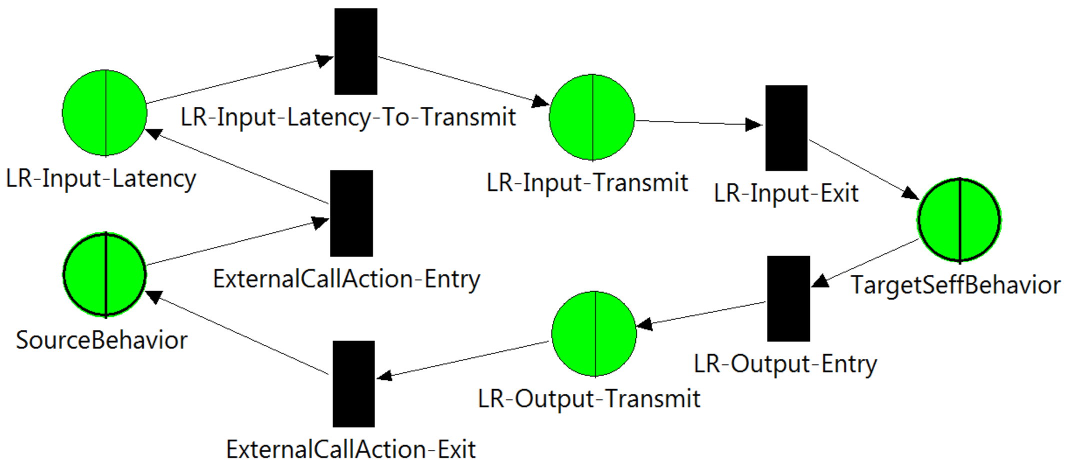


Figure 5.30: LinkingResource QPN

Using the information from the selected *LinkingResource* and the input and output parameters of the *ExternalCallAction*, the *ExternalCallAction* mapping is extended. This is shown in Figure 5.30. On the source side, the *ExternalCallAction-Entry* and *ExternalCallAction-Exit* transitions remain the same. On the target side, a number of new queueing places and transitions are generated. All transitions are simply connectors and consume one token in the source place and generate one token in the target place. The interesting part consists of the three new queueing places. *LR-Input-Transmit* and *LR-Output-Transmit* link to the same queue, which is generated on demand for each *LinkingResource*. It is a FCFS queue with 1 server. The resource demands are computed by evaluating the sum of the 'BYTESIZE' characterization of both all the input and all the output parameters. Each sum is then divided by the *throughput* of the *LinkingResource*. *LR-Input-Latency* behaves differently. Another queue is generated on demand per *LinkingResource* with 1 server and an Infinite Server (IS) scheduling strategy. As the resource demand the distribution of the *latency* specification is used.

This structure behaves as expected. An incoming token is first delayed with no contention in the delay place (as it is an Infinite Server (IS) queue) and then the load of its input parameter bytesize is placed on the transmit queue, which is shared between all tokens that represent requests between any of the containers connected by the *LinkingResource*. On the return to the caller, the bytesize of the output parameters is placed on the transmit queue again. This both delays the token and creates contention when the load increases.

The input and output parameters, as well as the *throughput* and *latency* specifications are handled by the DependencySolver (Section 5.1) and therefore general limitations apply to the use of stochastic expressions.

5.2.10 QoS Annotations

In the PCM meta-model, there are two types of quality-of-service annotations. A *SystemSpecifiedExecutionTime* is used to define the time that requests that target *RequiredRoles* of the *System* take to complete. A *ComponentSpecifiedExecutionTime* is used when one of the design-time component types (*ProvidesComponentType* or *CompleteComponentType*) are used in the system composition. As no behavior is specified for design-time components, the execution times of the provided services must be specified. This is only necessary when a design-time component is actually referenced by an *AssemblyContext*. All QoS annotations are contained in the *System*. The QoS annotations are only mandatory when the entities in question are targeted by an *EntryLevelSystemCall* or an *ExternalCallAction*.

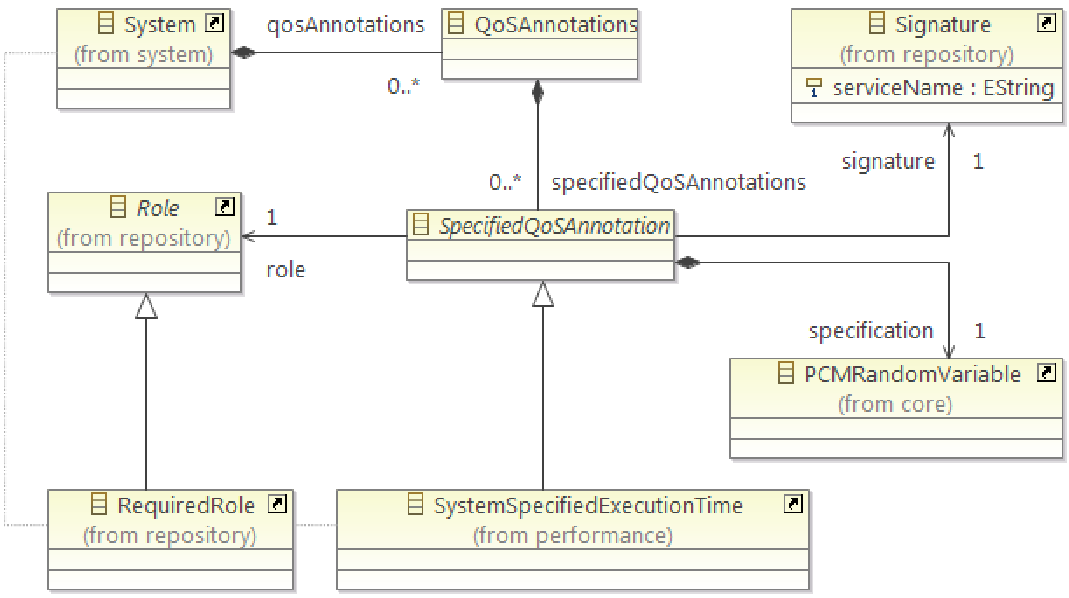


Figure 5.31: SystemSpecifiedExecutionTime PCM

Figure 5.31 shows the situation for system required services in more detail. The base entity *SpecifiedQoSAnnotation* of *SystemSpecifiedExecutionTime* references a *Role*. Generally a *Role* is used in any *ComposedStructure*. A detailed description of the composition hierarchy can be found in Section 5.2.3. For *SystemSpecifiedExecutionTimes* however, only the *RequiredRoles* of the *System* are valid targets. As the *Role* only specifies the interface, a reference to the target *Signature* is provided. The time a request is delayed when accessing one of the targeted services is specified in seconds in the *specification* property through a *PCMRandomVariable*.

In case of a *ComponentSpecifiedExecutionTime*, the situation is similar (Figure 5.32). One difference is that now the annotation does not always target the *System* and thus the target *AssemblyContext* is specified. Furthermore, different constraints apply. The target *Role* must now be a *ProvidedRole* and it must be contained in a *RepositoryComponent* of one of the design-time component types (*ProvidesComponentType* or *CompleteComponentType*).

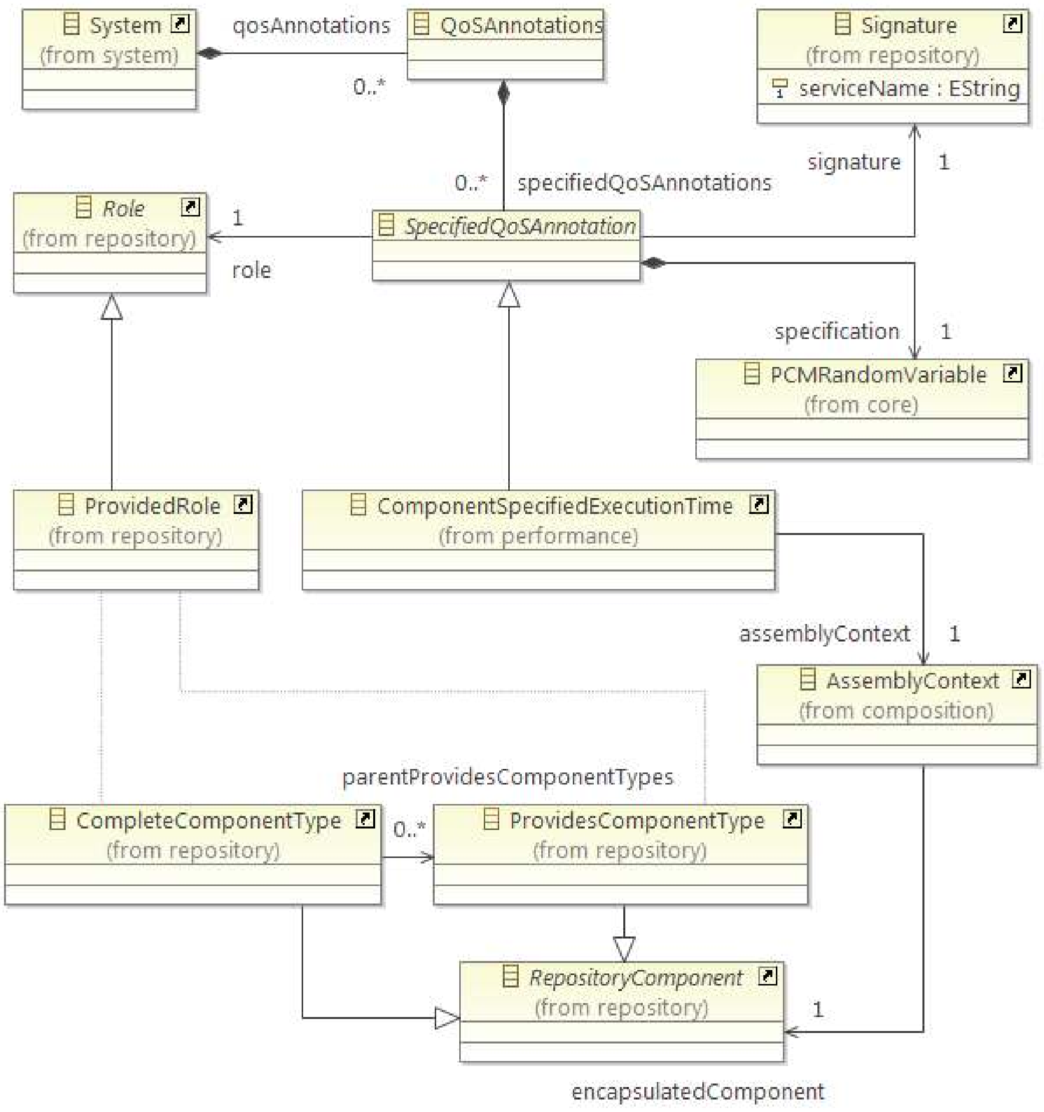


Figure 5.32: ComponentSpecifiedExecutionTime PCM

The QoS annotation information is queried by the mapping transformation whenever one of the calls (Section 5.2.3) cannot find a corresponding SEFF. To delay the tokens in the right place, the mapping for *InternalAction* (Section 5.2.7) is reused in both cases. A dummy *ProcessingResourceSpecification* with a DELAY scheduling strategy and a processing rate of 1 is generated on demand. Also, a dummy *InternalAction* is generated on demand with the execution time specification taken for the *ResourceDemand*. As the delay queue has no contention, all tokens are delayed exactly by the mean of the *specification* of the *SpecifiedQoSAnnotation*.

There is one implementation-specific issue regarding QoS annotations: the employed version of the DependencySolver module does not handle the use of input parameters in stochastic expressions for the execution time *specification*.

5.3 Simulation-related Mappings

5.3.1 Overview

The QPN representing the PCM instance is generated to analyze the PCM instance through simulation. To identify what is to be measured, a top-down approach is followed. First, a number of meaningful metrics for PCM elements are identified. Then, from different possibilities of calculating each metric, one is selected for which the information can easily be provided by the simulation. Afterwards, the mapping is extended with missing constructs only needed for the gathering of a metric and with the configuration information for the simulation. In a final step, the results integration code is written that aggregates the simulator data and computes from it the metrics in the PCM domain. This is necessary as the simulation data does not directly map to PCM metrics and the performance analyst is not expected to know the details of the transformation and simulation technologies.

5.3.2 Supported Metrics

The initial target was to at least support the following metrics which are also offered by the SimuCom simulator (although the throughput and mean response time have to be computed manually from the collected data):

- *UsageScenario* response time (distribution).
- *UsageScenario* mean throughput.
- *ProcessingResourceSpecification* mean total utilization (over all request types).

The following metrics are supported by the SimQPN solver. Where possible, the metrics are offered both per *UsageScenario* and in total. Per *UsageScenario* means accumulated over all requests that originate in that *UsageScenario*.

- Response time distribution of *UsageScenario*
- Mean response time per *UsageScenario* of
  - *UsageScenario*
  - *EntryLevelSystemCall*
  - *ExternalCallAction*
  - *ResourceDemandingBehavior* (includes *ServiceEffectSpecification* and *ForkedBehavior*)
  - *InternalAction*

- Mean throughput
  - *AbstractAction* (per *UsageScenario*)
  - *ProcessingResourceSpecification* (per *UsageScenario* and total)
  - *LinkingResource* (per *UsageScenario* and total)
- Mean utilization
  - *ProcessingResource* (per *UsageScenario* and total)
  - *PassiveResource* (per *AssemblyContext* over all *UsageScenario*s)
  - *LinkingResource* (per *UsageScenario* and total)

More metrics are technically possible but are out of the scope of this thesis and will be considered in future work:

- Response time distribution of *EntryLevelSystemCall*, *ExternalCallAction*, *ResourceDemandingBehavior*, *InternalAction*, *ScenarioBehavior*
- Breakdown of metrics per *EntryLevelSystemCall* and per *ExternalCallAction* (in addition to per *UsageScenario*).
- Mean throughput per *AbstractUserAction* (e.g., usage model *Loop*, *Branch*, *EntryLevelSystemCall*).
- Mean response time of *ScenarioBehavior* (e.g., usage model *Loop* or *Branch* child behaviors).
- Other metrics which can be derived from simulator data (e.g., min/max token population in a place).

5.3.3 Instrumentation Model

To extend the QPN in the right places, the mapping transformation needs to know the PCM elements that are to be instrumented and the metrics to collect. This is realized through a simple decorator model. In the PCM Solver tool implementation (Section 6.3) the ProbeSpec meta-model is used. The ProbeSpec meta-model has not yet been published and is likely to change. Therefore only the general idea of the decorator model is provided. There is a host entity similar to other EMF-based meta-models. It contains entities for each distinct metric to gather. In each entity the type of metric (e.g., mean response time, throughput, etc.) is stored and an EObject reference is kept that points to the PCM instance element for which that metric should be measured. In the following section, it is assumed that during the mapping the instrumentation information is available.

5.3.4 Instrumentation Mapping

This section deals with the actual mapping extension made to support gathering of the specified metrics. The SimQPN simulator supports specifying the data to be collected during the simulation run for each place separately. For each place (ordinary and queueing place), and for each probes measurement instance (see Section 2.2.5 for an introduction to probes), a stats-level is configured which determines the data that is collected. The higher the level the more data is collected at the cost of simulation time. Only the levels and the data we use in the mapping are presented here. More information can be found in [KD09].

The following stats-levels and corresponding QPN metrics are used:

stats-level 0 No measurements (and no wasted execution time).

stats-level 1 Mean throughput only. *deptThrPut* is the mean throughput available per token color. *totDeptThrPut* is the mean token throughput over all colors.

stats-level 2 Mean token population at the place (for ordinary places) or at the depository (for queueing places). *meanTkPop* is the mean token population per token color. *meanTotTkPop* is the mean token population over all colors. Total queue utilization at the queue referenced by a queueing place (*queueUtil*). The other metrics measured at this stats-level remain unused.

stats-level 3 Token residence time data. In this case we use the mean residence time (sojourn time) *meanST* for probes only. The other data entries like the minimum and maximum residence time, as well as a confidence interval are not used.

stats-level 4 Histogram for token residence times. A bucket size is specified at the start of the simulation. The number of resulting buckets and the number of tokens in each bucket are provided.

The throughput PCM metrics use a stats-level 1 configuration at the start place of the QPN subnet that was generated for the target element. It is assumed that during the simulation a steady state is reached so that the throughputs in the start place and in the end place are equal. The mean throughput that corresponds to a given *UsageScenario* for that PCM element is then the sum over all *deptThrPut* entries of the colors that belong to that *UsageScenario*. The total throughput is *totDeptThrPut*.

The PCM utilization metrics for a *ProcessingResourceSpecification* and for a *LinkingResource* use a stats-level 2 configuration in all places P_i that reference the underlying queue (transmit queue in the case of *LinkingResource*s). The mean utilization per *UsageScenario* is calculated using the multiclass version of the Utilization Law. It states that the mean utilization $U_{i,r}$ of resource i and class r equals the mean service time $S_{i,r}$ multiplied by the mean throughput $X_{i,r}$ (at that resource i and for that class r). The resource i is fixed and represents the measured queue. There is one class r for each pair of place P_i and token color at the place. The mean throughput $X_{i,r}$ then equals *deptThrPut* for a color and place combination representing r. The mean service time $S_{i,r}$ is computed during the mapping and is available. The mean utilization per *UsageScenario* is now the sum over all $U_{i,r}$ of all classes r that correspond to that *UsageScenario* for the measured queue i. The total utilization is measured at the queue itself and equals *queueUtil*.

The utilization for a *PassiveResource* is computed differently. There is one ordinary place for each instance of the *PassiveResource* that corresponds to an *AssemblyContext*. No further distinction per *UsageScenario* is made because a passive resource could be acquired by the request of one *UsageScenario* and released by the request of another. Therefore all places representing *PassiveResource*s can use only one type of color. The utilization per *AssemblyContext* can be computed using the *meanTotTkPop* P of the corresponding place and the initial capacity C of the *PassiveResource*. To make *meanTotoTkPop* available, stats-level 2 is configured for the place. The utilization then equals $(C - P)/C$. When no requests have acquired the resource, the mean token population P is still at the initial capacity C and the utilization equals $(C - C)/C = 0.0$. If all available resources are acquired, no tokens remain and $P = 0$. The utilization then equals $(C - 0)/C = 1.0$.

The PCM mean response time metrics can be computed with two different methods. One possibility is to compute the mean of the response time distribution measured using probes. If only the mean response time is needed, a more efficient method can be used. The method is selected through a configuration option. The probes option is currently only available for *UsageScenario*s. All other mean response times are calculated using the optimized method.

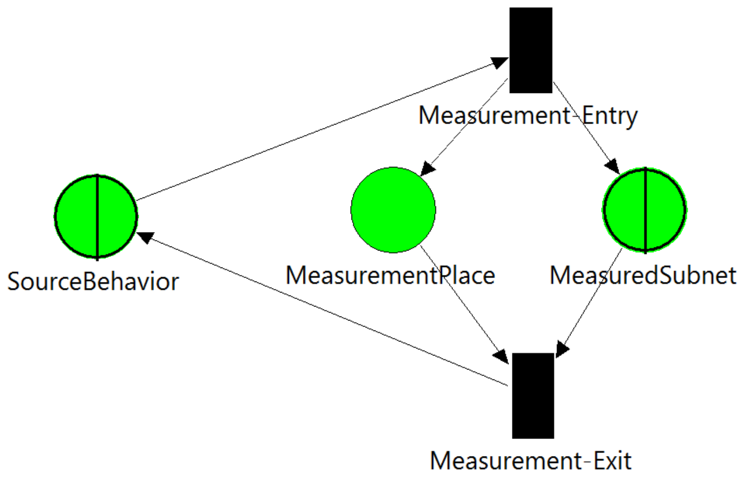


Figure 5.33: Measurement Place QPN

The optimized method of computing the mean response time requires a special measurement place, illustrated in Figure 5.33. The measurement place is connected in parallel to the subnet for which the mean response time is to be measured. For each token of color c that is created in the subnet by the *Measurement-Entry* transition, a token of color c is created in *MeasurementPlace*. Likewise, for each token that is consumed from the subnet one is consumed from the *MeasurementPlace* as well. The *MeasurementPlace* always contains the same number of tokens of color c as the whole subnet. The *MeasurementPlace* is configured with stats-level 2 and the mean response time is then calculated using Little's Law. Little's Law states that the average number N of tokens in a black-box system equals the average departure (or arrival) throughput X times the average residence time R. Treating each color separately this becomes $N_c = X_c * R_c$ for each color c. The mean response time for a color c corresponds to the mean residence time R_c and equals N_c/X_c. As several token colors can correspond to a single *UsageScenario*, we use Little's Law for a new combined color k that represents all of the colors of that *UsageScenario*. R_k is now N_k/X_k. N_k is now the sum over the individual token populations *meanTkPop* of all the colors combined in k. Likewise, X_c is the sum over all throughputs *deptThrPut* of the same colors. The final division N_k/X_k completes the computation of the mean response time of that subnet for the given *UsageScenario*.

The *UsageScenario* response time distribution is implemented using probes. The first ordinary place of the usage scenario (*UsageScenario-Entry*) is marked as the starting place of the probe. Measurements are set to start on the entry of the tokens. Likewise, the *UsageScenario-Exit* place is marked as the ending place of the probe with measurements to be taken on exit of the place. Tokens are annotated with an id and timestamp pair in addition to their color. Response time statistics become available for the whole marked subnet. As the PCM request is represented by a number of colors during its journey, each of the colors is added to a list of colors for which the timestamp is to be passed on. For other colors it is dropped. As the basic Queueing Petri Net behavior is not modified (the id annotation is non intrusive), an approximation occurs whenever two or more tokens of colors marked to carry timestamps are needed for a transition to fire. Instead of waiting for tokens of an equal timestamp (and thus the completion of the whole request), a random timestamp pair is selected. This approximation comes into effect for loops (see Section 5.2.5) synchronous forked behaviors (see Section 5.2.6) and when *MeasurementPlace*s are generated.

6. PCM Solver Tool

This chapter covers the solving process, especially the PCM-to-QPN mapping transformation, from a more practical point of view. First the architecture of the PCM Solver tool is presented in Section 6.1. Section 6.2 covers the most important modules and the basic control flow of the tool in more detail. An overview of the QVTO implementation and Java blackbox integration is presented in Section 6.3. Finally, Section 6.4 discusses limitations and issues that will be addressed in future work.

6.1 Architecture

All software artifacts that are developed and used in this thesis are packaged as Eclipse plugins. Since there are many individual plugins, we do not describe them separately. Instead, related plugins are grouped and given a simpler name. Those groups make up the major components of the architecture of the tool. The next section on design (Section 6.2) will look at these components in more detail. In this section, an overview of the major parts is provided. The most important part, the *PCM-2-QPN* plugin, will also be presented here in more detail. It implements most of the mapping transformation described in Chapter 5.

Figure 6.1 shows the simplified plugin architecture. All plugins run inside the *Eclipse runtime* environment. *PCM Core* denotes all plugins supplied through the PCM update site, except of the PCM-Bench. *PCM Core* includes the EMF meta-model of PCM and the PCM Workflow Engine. The *PCM-Bench* is shown separately because it provides the user interface of PCM. The *instrumentation user interface* refers to the part of the user interface that is used to specify the instrumentation decorator model described in Section 5.3.3. Currently it is realized as a very simple EMF editor. A future version of the PCM bench is likely to include a more sophisticated version this tool will make use of (Section 9.2). As the DependencySolver updates only the stochastic expressions of the existing model, the decorator model is valid for the *Solved PCM Instance* as well. The launch configuration handlers and the transformation itself are contained in the *PCM-2-QPN* plugin. More information on user interaction can be found in Section 3.2.1. The transformation targets SimQPN, which is part of QPME (Section 2.2.5). QPME is divided into two separate plugins, QPE and SimQPN. *QPE* covers the editor related parts and *SimQPN* the simulation related parts. QPME is not based on EMF but uses XML files for input and output. Therefore EMF meta-models both for the *QPE* serialization format, which is the input format of *SimQPN*, as well as for the simulation output of *SimQPN* were generated. They were generated from XML schemas derived from a number of available

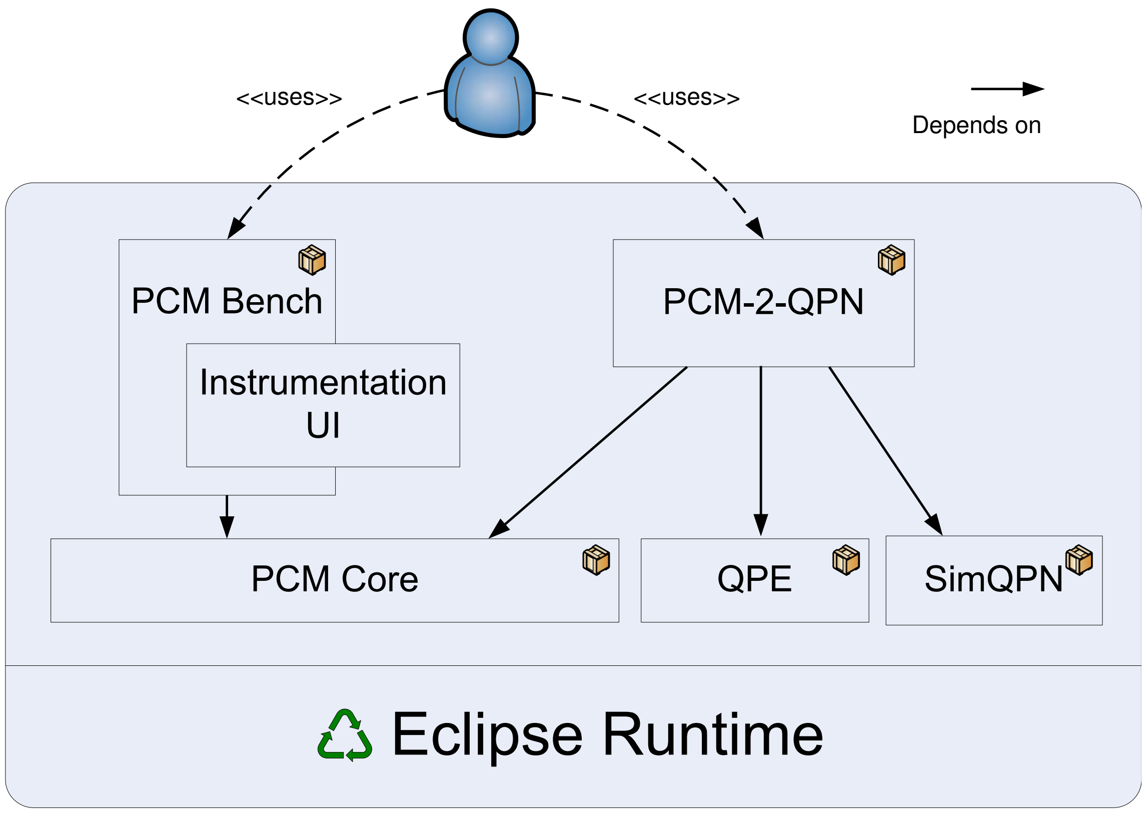


Figure 6.1: Simplified Plugin Architecture

sample files. The meta-model projects have been maintained manually from that point on. Serialization annotation ensures that the XMI serializer follows the format *SimQPN* expects.

Figure 6.2 shows the architecture of the *PCM-2-QPN* plugin and the flow of the transformation artifacts. The transformation is built on top of the *PCM Workflow Engine* which provides common functionality like loading a PCM instance into memory and executing a QVTO transformation script on it. There are four major jobs: The *Dependency Solver Job* executes the DependencySolver on the provided PCM instance, solving the contained parametric dependencies. This is described in Section 5.1. The *PCM-2-QPN Transformation Job* implements the mapping transformation, taking a PCM instance with solved dependencies (denoted as Solved PCM Instance in the figure) and the instrumentation decorator model to generate a matching QPN model. The *SimQPN Simulation Job* executes the SimQPN simulator on the generated QPN model, creating an instance of the SimQPN results meta-model. The *SimQPN Solver Job* acts as a mediator for the other jobs. It is also the target of the launch configuration of the user.

The SimQPN Solver expects from the user a valid and complete PCM instance, as well as an instance of the instrumentation decorator model. After running the solver, the Results Integration extension is used to feed back the results to the user in a meaningful way (see Section 5.3.4). Currently the results are printed to the console and to a results file.

6.2 Design

6.2.1 Modules

Figure 6.3 shows the most important modules of the solver tool and the plugins they reside in. Most modules are located in the *edu.kit.ipd.sdq.simqpn.solver* plugin, which

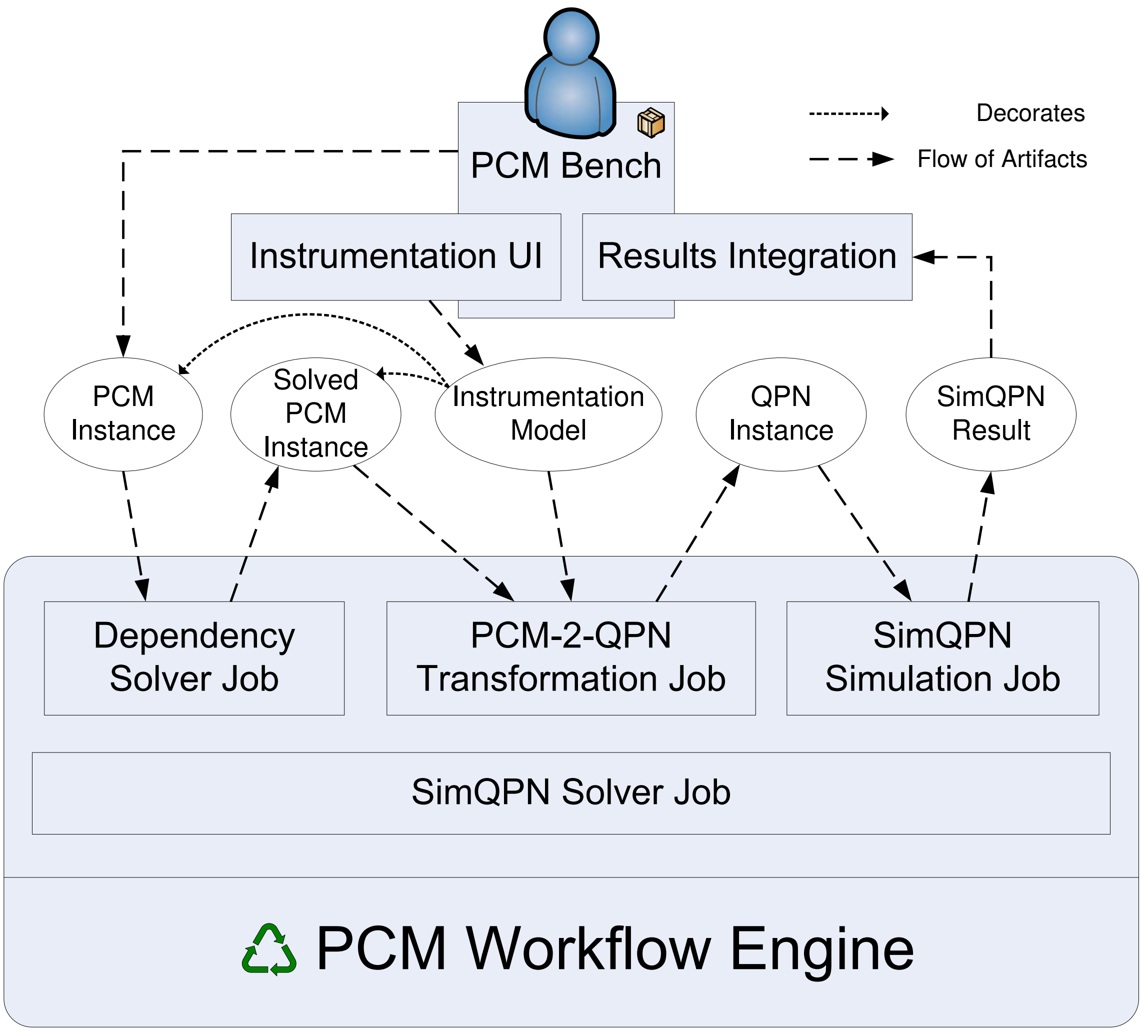


Figure 6.2: PCM-2-QPN Architecture

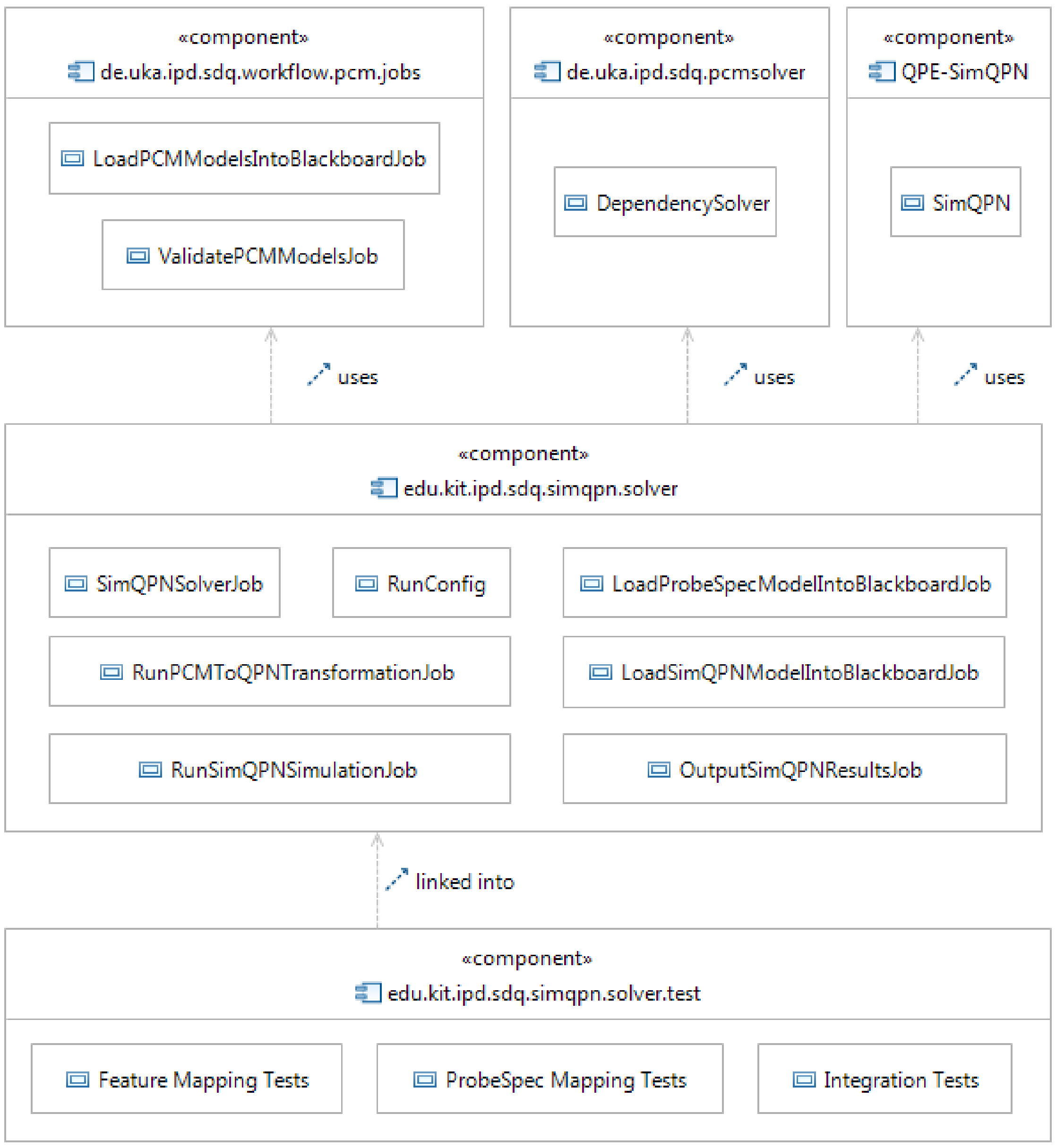


Figure 6.3: Solver Tool Modules

was developed throughout this thesis. As mentioned in the last section on architecture, the *SimQPNSolverJob* acts as mediator for all steps that are needed to solve the given PCM instance. *RunConfig* contains all classes needed to handle the user interaction and serialized launch configuration. A new launch configuration type is defined and a simple interface similar to what the SimuCom simulator offers is implemented. The other modules in *edu.kit.ipd.sdq.simqpn.solver* each implement one step of the solving process. The substeps of loading a PCM instance into memory and validating it are handled by modules that are part of PCM (*de.uka.ipd.sdq.workflow.pcm.jobs*). The DependencySolver discussed in Section 5.1 is contained in *de.uka.ipd.sdq.pcmsolver*. All test classes and fixtures are located inside a test fragment *edu.kit.ipd.sdq.simqpn.solver.test* which references *edu.kit.ipd.sdq.simqpn.solver.test*. Classes within a fragment have package access to classes of its host plugin, but classes inside the host plugin cannot see classes in the fragment. This is an ideal setup for automated tests using JUnit. There are three classes of tests. The *Feature Mapping Tests* run the *RunPCMtoQPNTransformationJob* on small PCM instances that make use of individual features and check the resulting QPN mapping. The *ProbeSpec Mapping Tests* additionally load a ProbeSpec model instance (see Section 5.3.3) and check the instrumentation part of the mapping. The *Integration Tests* use PCM instances that use a number of elements in combination and a matching ProbeSpec instance. The whole solver process (the *SimQPNSolverJob*) is run and the results are checked regarding the specified metrics.

6.2.2 Control Flow

Figure 6.4 shows the control flow inside the *SimQPNSolverJob* module. The solver process takes a PCM instance, an optional ProbeSpec instance and the solver configuration as an input and emits the selected metrics to the console and to an output file. Currently the output file and all temporary files are put inside the users temporary system folder. If no ProbeSpec model is specified, only the mean response time and mean throughput of all *UsageScenario*s, as well as the utilization of all possible resource types (processing, passive and linking resources) are measured. This is handled by the *LoadProbeSpecIntoBlackboardJob* by creating a ProbeSpec model on-the-fly.

On the left side of the figure, we can see the *Blackboard*. The *Blackboard* is another component offered by PCM. It manages a set of partitions in memory which hold EMF models. This allows to decouple any details on loading the models from the jobs that actually work with the instances. The *RunPCMtoQPNTransformationJob* and the *OutputSimQPNTransformationJob* simply expect a valid model loaded in the blackboard. The SimQPN simulator expects a model in the form of an XML file. It also puts the results in an XML file. Therefore the blackboard is not used in this case and the result file has to be loaded onto the blackboard before the *OutputSimQPNTransformationJob* can access it.

One other file, omitted from the figure for simplicity, contains information on which PCM element was mapped to which places in the QPN. This information is saved by the *RunPCMtoQPNTransformationJob* and is then used by the *OutputSimQPNTransformationJob* to aggregate the result metrics. This information is stored in a set of Java hash maps that are serialized using the built-in mechanisms (with no need to create a custom meta-model). The class that manages this mapping file is called *MeasuredObjectToPlaceMapper*.

6.3 Implementation

The largest and most interesting part of the PCM Solver tool implementation is the QVTO transformation. QVTO is introduced in Section 2.3.2, details can be found in [OMG08a]. We will look at how the transformation is defined in the *pcm2qpn.qvto* script. To give the reader insight into how the transformation is implemented and how a typical code section

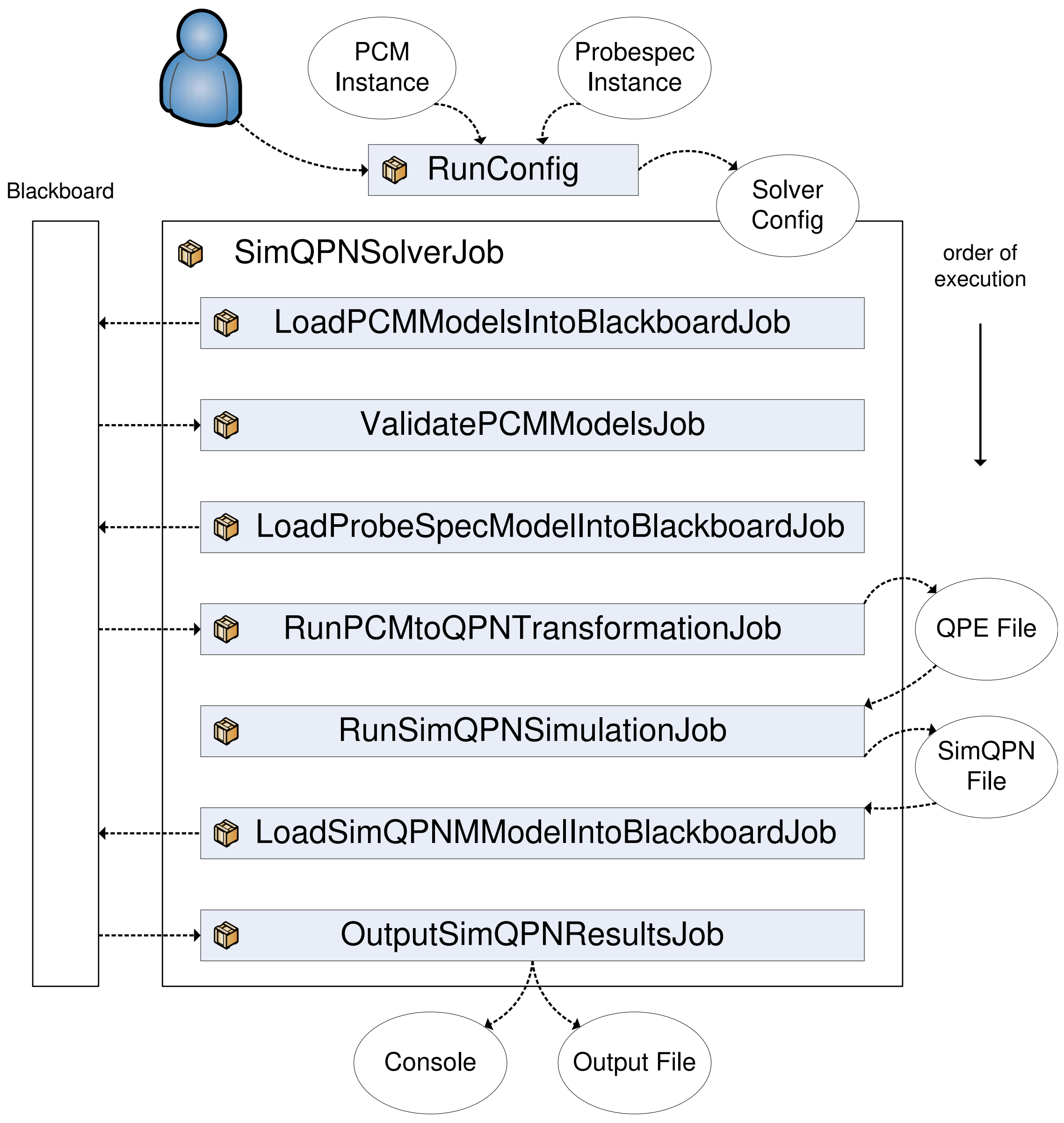


Figure 6.4: Solver Tool Control Flow

looks like, a simplified mapping and a couple of global helpers are introduced. Finally, we look at how the DependencySolver is integrated using a Java blackbox component.

6.3.1 QVTO Transformation

```
import edu.kit.ipd.sdq.simqpnsolver.qvto.IdManagerBlackBox;
import edu.kit.ipd.sdq.simqpnsolver.qvto.ContextWrapperBlackBox;
import edu.kit.ipd.sdq.simqpnsolver.qvto.
   MeasuredObjectToPlaceMapperBlackBox;
import PcmHelper;
import QpeHelper;
import ProbeSpecHelper;
import AssertHelper;

modeltype PCM_ALLOCATION uses
   "http://sdq.ipd.uka.de/PalladioComponentModel/Allocation/4.0";
modeltype PCM_USAGE_MODEL uses
   "http://sdq.ipd.uka.de/PalladioComponentModel/UsageModel/4.0";
modeltype PCM_REPOSITORY uses
   "http://sdq.ipd.uka.de/PalladioComponentModel/Repository/4.0";
modeltype PCM_SEFF uses
   "http://sdq.ipd.uka.de/PalladioComponentModel/SEFF/4.0";
modeltype PCM_SEFF_PERFORMANCE
 uses "http://sdq.ipd.uka.de/PalladioComponentModel/SEFF/Performance/1.0";
modeltype PROBE_SPEC uses "http://sdq.ipd.uka.de/ProbeSpec/0.1";
modeltype QPE uses "qpe_meta";
<...>

transformation pcm2qpn(in inAll : PCM_ALLOCATION,
   in inUsg : PCM_USAGE_MODEL, in inProbeSpec : PROBE_SPEC,
   out outputNet : QPE)
   access library IdManagerBlackBox, PcmHelper,
      ContextWrapperBlackBox, QpeHelper, ProbeSpecHelper,
      MeasuredObjectToPlaceMapperBlackBox, AssertHelper;
```

Figure 6.5: *pcm2qpn.qvto* Transformation Definition

Figure 6.5 shows the transformation definition part of the *pcm2qpn.qvto* script. Helpers which could be useful in other contexts were extracted to a number of libraries, which are imported by the transformation. This includes a helper for PCM meta-model elements, for elements of the QPE result type and for ProbeSpec elements. A number of helpers are realized as Java blackbox components. They use the *edu.kit.ipd.sdq.simqpnsolver.qvto* namespace in the import statement.

The figure shows that the pcm2qpn transformation has three input models and one output model. We take a model of type *PCM_ALLOCATION* with a root element of *Allocation*, a model of type *PCM_USAGE_MODEL* with a root element of type *UsageModel* and a model of type *PROBE_SPEC* with a root element of type *ProbeSpecRepository*. One output model of type *QPE* with a root element of type *DocumentRoot* is returned. It is important that more than the modeltypes for the input and output parameters are declared. As we use almost all entities of the PCM meta-model in the transformation, the remaining PCM sub-packages are also included as modeltypes.

```
// optional. emits warning if not set
configuration property measuredObjectToPlaceMapFileUri : String;

// optional property. default is false
configuration property isFatalErrors : Boolean;

// optional property. default is true
configuration property isProcessLinkingResources : Boolean;

configuration property simSettings_outputDirectory : String;
configuration property simSettings_rampUpLength : Real;
configuration property simSettings_totalRunLength : Real;

intermediate property pcm2qpe::resultNet : NetType;
intermediate property pcm2qpe::probeSpec : ProbeSpecRepository;
```

Figure 6.6: *pcm2qpn.qvto* Transformation Properties

Figure 6.6 shows the properties section of the pcm2qpn transformation. Most properties are configuration properties. One example is the Boolean property *isProcessLinkingResources*, which determines wether or not *LinkingResource*s should be considered when mapping *ExternalCallAction*s. Simulator settings stored in the result file are also passed in as configuration parameters. The *probeSpec* intermediate property is used only for convenience. It is set early in the $main()$ mapping. The $resultNet$ intermediate property is also initialized early in the $main()$ mapping. As we will see in the next example, the $resultNet$ is accessed throughout the transformation. QPN elements (places, transitions, queues, colors, connections) are added through helpers.

Figure 6.7 presents a coherent example of the mapping *CreateStructure* for the type *UsageScenario*. We can see how the mapping is invoked for each of the *UsageScenario*s in the *CreateStructure* mapping for the type *UsageModel*. The *UsageModel* mapping is invoked in the *createQueueStructure* helper. *createQueueStructure* is called directly from the $main()$ mapping, which has been simplified for this demonstration. We see the use of the built-in log helper, the initialization of the result net using another helper, as well as how after *createQueueStructure* returns, the result root object of type *DocumentRoot* is created. Its net property is initialized with the $resultNet$ intermediate property.

In $UsageScenario :: CreateStructure$ we see a section of code which is typical for the whole transformation. The id helper is used to generate fresh ids (using $newId()$) for QPN elements that are about to be created. The id is stored in a temporary variable (e.g., $clientColorId$). Then global add-helpers are used to add QPN elements to the result net. They are created using the *new* keyword and a constructor. In a second step more information (e.g., color references) are added to the newly created QPN elements. The element is retrieved by its id and another helper is used to set the required information. Finally, other mappings are called for each of the client entities. In this case one mapping is executed for the *Workload* and for the *ScenarioBehavior* of each *UsageScenario*. *Workload* is an abstract entity. In this case an abstract mapping is declared for the type *Workload*, which defines a common parameter signature. For each of the concrete subclasses another mapping with the same signature is created. The QVTO interpreter chooses the most specialized mapping at runtime.

The mapping signatures in the pcm2qpn transformation generally are passed an entry place id, an exit place id, the current request color, and a context reference that can be

```
main() {
   log("Starting pcm2qpe transformation");

   processConfigurationParameters();
   initResultNet();
   <...>

   createQueueStructure();

   <...>
   // return results net
   object DocumentRoot {
      net := this.resultNet;
   };
}

helper createQueueStructure() {
   inUsg.getUsageModel().map CreateStructure();
}

mapping UsageModel::CreateStructure() {
   self.usageScenario_UsageModel.map CreateStructure();
}

mapping UsageScenario::CreateStructure() {
   var clientColorId : Integer := newId();
   var colorName : String := "client_" + self.id;

   // create client token
   addColor(new ColorType(colorName, clientColorId));

   //create usage model entry and exit places
   var usageModelEntryPlaceId : Integer := newId();
   var usageModelExitPlaceId : Integer := newId();
   addPlace(new PlaceType("UsageScenario-Entry_" +
      self.id, usageModelEntryPlaceId, "NORMAL"));
   addPlace(new PlaceType("UsageScenario-Exit_" +
      self.id, usageModelExitPlaceId, "NORMAL"));

   getPlaceById(usageModelEntryPlaceId).addSimpleColorRef(clientColorId);
   getPlaceById(usageModelExitPlaceId).addSimpleColorRef(clientColorId);

   self.workload_UsageScenario.map CreateStructure(usageModelEntryPlaceId,
      usageModelExitPlaceId, clientColorId);

   self.scenarioBehavior_UsageScenario.
      map CreateStructure(usageModelEntryPlaceId,
         usageModelExitPlaceId, clientColorId);
}
```

Figure 6.7: *pcm2qpn.qvto* Mapping Example

used to retrieve information from the DependencySolver. In this case we are still traversing the *UsageModel* and no context is required.

```
helper addPlace(place : PlaceType) {
   this.resultNet.places.place += place;

   return;
}


query getPassiveResourcePlaceId(assCtx : AssemblyContext,
   res : PassiveResource) : Integer {
   var key: String := "place_passive_resource_" +
      assCtx.id  + "_" + res.id;

   return getId(key);
}
```

Figure 6.8: *pcm2qpn.qvto* Helper Examples

To finish our example, we will look at two helpers which represent two common classes of helpers (Figure 6.8). *addPlace* is one of the *add*-helpers used to add QPN elements to the result net. We can see that the intermediate property *resultNet* is accessed using the *this* keyword and that the place passed as a parameter is added to the *places* collection using the *+=* operator. The *places* collection is referenced using *places.place* because the XSLT to EMF converter has introduced a special collection modeltype *PlacesType* in the QPN meta-model.

getPassiveResourcePlaceId represents another common class of helpers. They are used to retrieve the id of a QPN element which is uniquely determined by a set of PCM elements. These helpers are used when one QPN entity needs to be accessed from different parts of the transformation. A unique string is built from input elements and another id helper *getId*() is used to retrieve the id. The first time *getId*() is accessed, a new id is generated.

One situation in which a QPN element is accessed from different parts of the mapping is when the mapping for an element is split up into two parts. One part creates the "structure" for that element. The "structure" in this case means the places, transitions and connections that are shared by all mappings for that particular PCM element. An example is the pool place of a *LoopAction*. Depending on the context, the mapping for the *LoopAction* with the same id is executed a number of times with different context parameters. The "structure" is used between all mappings and is generated only once. The other part of the mapping is then called "color structure". This is the context dependent part (e.g., the color ids) which differts for each execution. The ids of the elements created in the "structure" part of the mapping are then retrieved using a matching *getId*-helper.

It is noteworthy that due to traversing PCM elements several times with different context information, features like result element tracing and other more advanced QVTO features are not used in the pcm2qpn transformation.

6.3.2 Java Integration

In this section, the integration of the DependencySolver is presented as a concrete example of a Java blackbox component. Figure 6.9 shows the extension point definition in the *plugin.xml* of the *edu.kit.ipd.sdq.simqpnsolver* plugin. The namespace, which is shared by all our blackbox components is *edu.kit.ipd.sdq.simqpnsolver.qvto*. The name of the

```
<extension point="org.eclipse.m2m.qvt.oml.javaBlackboxUnits">
   <unit name="ContextWrapperBlackBox"
      namespace="edu.kit.ipd.sdq.simqpnsolver.qvto">
   <library name="ContextWrapperBlackBox"
         class="edu.kit.ipd.sdq.simqpnsolver.qvto.ContextWrapperBlackBox">
   <metamodel
      nsURI="http://sdq.ipd.uka.de/PalladioComponentModel/UsageModel/4.0">
   </metamodel>
   <metamodel
      nsURI="http://sdq.ipd.uka.de/PalladioComponentModel/SEFF/4.0">
   </metamodel>
   <...>
   </library>
   </unit>
</extension>
```

Figure 6.9: DependencySolver Blackbox Definition

DependencySolver integration library is *ContextWrapperBlackBox*. The name comes from the main class of the DependencySolver, the *ContextWrapper*. The PCM usage model and SEFF packages are loaded amongst others. PCM elements like *UsageScenario* can therefore be used in the signatures of methods that are annotated using @*Operation*.

Figure 6.10 shows the Java class implementing the blackbox component. One of the exported operations is shown. *getContextWrapperFor*() takes an *EntryLevelSystemCall* and returns a reference to a *ContextWrapper* class. Because the *ContextWrapper* class is a plain Java class and not a class representing an EMF meta-model entity, a generic *Object* reference is returned. This turns into the QVTO datatype *OclAny*. As we can see, *getContextWrapperFor()* requires an initialized *rootWrapper*. The *rootWrapper* is a static variable of the *ContextWrapperBlackBox* class. It is static so it can be initialized with the result of the DependencySolver preprocessing step (a *PCMInstance*) before the QVTO interpreter runs the *pcm2qpn.qvto* script.

6.4 Limitations

One limitation regarding the code reusability of the PCM Solver tool is that not all new library code for traversing and managing a PCM instance was placed in the DependencySolver module. Some helpers were written in QVTO or were placed inside the DependencySolver blackbox component class and are not available to other projects that rely on the DependencySolver. The library methods could, however, be refactored into the DependencySolver.

The DependencySolver module comes with its own Eclipse launch configuration classes and a strategy pattern to include new solvers that build on the DependencySolver. The existing architecture did, however, not consider the use of the PCM workflow engine, especially the blackboard needed for the QVTO execution. To simplify the building of the initial prototype of the PCM Solver tool, a separate launch configuration was used.

```
package edu.kit.ipd.sdq.simqpnsolver.qvto;

<imports>

public class ContextWrapperBlackBox {

   private static ContextWrapper rootWrapper = null;

   public static void setPcmInstance(Object pcmInstance) {
      rootWrapper = new ContextWrapper((PCMInstance) pcmInstance);
   }

   private ContextWrapper castWrapper(Object contextWrapper) {
      return (ContextWrapper) contextWrapper;
   }

   @Operation
   public Object getContextWrapperFor(EntryLevelSystemCall elsa) {
      if (rootWrapper != null) {
         try {
            return rootWrapper.getContextWrapperFor(elsa);
         } catch (Exception e) {
            // do nothing
         }
      }
      return null;
   }

   <...>
}
```

Figure 6.10: DependencySolver Blackbox Class

7. Evaluation

In this section, we present a detailed evaluation of the PCM-to-QPN mapping. Section 7.1 shows how the metrics used for the evaluation were derived using the Goal/Question/Metric approach [BCR94]. Section 7.2 lists the assumptions of the evaluation. Section 7.3 describes the runtime environment and hardware setup of the evaluation runs. Section 7.4 presents the evaluation steps and the results of the feature support evaluation. The case studies are presented in their own chapter (Chapter 8).

7.1 Goal - Question - Metric

The main goal of this diploma thesis is to *evaluate the usefulness of QPNs as target analysis model for PCM.* This main goal implies several subgoals. One goal is semantical correctness of the mapping. In cases, where the exact semantics were unclear, the behavior of the SimuCom solver was taken as a guideline. From this goal the following question is derived: *Do the feature mappings behave as expected during simulation time?* From this question, the requirements for the feature support evaluation metrics are derived:

- The metrics must be collected for each feature separately.
- The metrics must relate to the correct semantics of the feature in question.
- All compared solvers must support collection of the metrics.

These requirements were used to derive the feature support evaluation steps described in Section 7.4.1. It consists of evaluation models for each feature and a small set of manually derived reference metrics. These scenarios are very similar to integration tests. The accuracy of the feature mappings can only be checked for a fixed number of cases. As the time for this diploma thesis is limited and the number of features is large, not all variations of the features could be covered. Additional scenarios, especially scenarios that use more advanced stochastic expressions and parameter passing, would be needed to reach a thorough understanding of the feature support.

Knowledge about which features are not supported or about features that cause crashes is still very useful. The feature scenarios are therefore also part of the evaluation to determine *when to use each solver.* They are used to answer the derived question: *Which features are supported by each solver?*

A second set of important questions relate to the usability of each solver in real world situations. The following questions cannot be answered using small, arbitrarily created, feature models: *What is the accuracy of the mapping for a representative system? Can the solver handle larger models?* These questions lead to different requirements for the metrics:

- The metrics must relate to the real world complexity of a system.
- The metrics must allow to evaluate the precision of the analysis.
- It must be possible to link the metrics to the real execution time of the analysis.
- All compared solvers must support the collection of the metrics.

The solution to these requirements is to conduct a number of case studies in which external PCM instances are analyzed that were modeled after systems of realistic size and complexity. The evaluation steps for the case studies are described in Section 8.2.

SimuCom offers a relative precision stopping criterion for the response time of usage scenarios. This requires the measurement of the response times of individual requests. With limitations, this is possible in SimQPN through the probes feature. The feature was added late during the thesis and was not available for the major part of the evaluation. Instead of using a relative precision stopping criterion for all compared solvers, a suitable fixed simulation time is determined individually for each case study using SimuCom.

The evaluation of the usability and portability of the different solver tools is out of the scope of this thesis and will be considered in future work.

7.2 Assumptions

A major assumption of the evaluation is that the DependencySolver handles parametric dependencies and stochastic expressions correctly.

It is also assumed that the employed models do not rely on a middleware overhead model, which is only supported by SimuCom and out of the scope of this thesis.

Another assumption is that a trained performance analyst is able to determine a suitable fixed simulation time to analyze a system. For the case studies, the relative precision stopping criterion of SimuCom for the mean response time of a selected *UsageScenario* is used. A direct support in the solver tool developed for this thesis will be addressed in future work.

7.3 Experimental Environment

All experiments were conducted using the Eclipse environment described in Section 3.2.2. The operating system and hardware environment were as follows:

- Microsoft Windows 7 Professional (64bit)
- 32bit JDK 1.6.0 (Update 20)
- Quad-Core Intel i5 750 CPU (2.67 Ghz per core)
- 4 GB Memory. Substantially less was available for a simulation run due to the 32bit JDK, which was chosen due to compatibility reasons.

The solvers were available in the following versions:

- PCM 3.2 Development Build
- QPME 1.5.2 Development Build
- LQNS and LQSim 4.1

Of the available simulation methods offered by SimQPN, the batch means method was used for all simulation runs. It is the most stable method. The batch means method was also used for the LQSim solver, as it is the standard method of operation. The exact command line call for LQSim used for the evaluation is:

```
lqsim -T<logical runtime> -o<output filename> <input filename>
```

LQNS is called with:

```
lqns -o<output filename> <input filename>
```

Additionally, the following LQNS settings are generated into the input file: $convValue = 1e - 005$, $itLimit = 50$, $printInt = 10$, $underCoeff = 0.5$ and $psQuantum = 0.001$. These are the default settings chosen by the PCM-to-LQN transformation.

7.4 Feature Evaluation

In this section, the first part of the evaluation, the evaluation of the feature support, is presented. Before discussing the results, the steps that were executed to conduct the evaluation are presented.

7.4.1 Steps

To analyze which solvers support which features, the following steps are taken for each of the mapped features:

1. Create a minimal, but complete model that uses the feature to be evaluated.
2. Choose from metrics which are available for all solvers (which turned out to be *UsageScenario* mean response time and throughput, *ProcessingResourceSpecification* utilization) at least one which can be derived by means of exact analytical analysis.
3. For each of the chosen metrics derive the results analytically.
4. Analyze the model 30 times with each solver, deriving the respective metrics.

The simulation results are then analyzed using R and a feature support table is derived. A 95% confidence interval for the mean of each metric over the 30 measurements is computed. The maximum relative error with respect to the expected result of all metrics and for both the left and right border of each interval is computed. If that maximum error is $\leq 5\%$, the feature is considered supported, otherwise, the feature is considered not supported. A crash in any of the 30 runs for a feature is considered as no support. Unexpected behavior or anomalies are noted. A similar evaluation approach based on the relative error between two measurements can be found in [Tri10].

7.4.2 Results

It is important to point out that during the time available for the thesis, the DependencySolver module, which the SimQPN, LQSim and LQNS solvers build on, was extended to support some of the less common features. There was no time to update the PCM-to-LQN transformation and those features cause problems or are not supported with the LQSim and LQNS solvers. It is out of the scope of this thesis to determine which of the unsupported features could be mapped to LQNs and with which limitations. This section is therefore only meaningful for the versions of the solvers used for this evaluation. The infrastructure of the feature support evaluation can be reused once new versions of the solvers become available to guide the user in choosing the right solver for the model at hand.

Table 7.1 shows the results of the feature support evaluation. The *Feature* column contains the name of the evaluated feature. On the right side we find one column for each of the examined solvers. At least one metric that is dependent on the feature semantics is manually derived and taken as a reference value. The exact models and derivation are omitted here for reasons of brevity but are available on request. Supported features are marked with S, unsupported features with N. A crash during any of the runs is marked with C. At the bottom of the table the number of supported features, not supported features and crashes are listed.

Feature	**LQNS**	**LQSim**	**SimQPN**	**SimuCom**
OpenWorkload	S	S	S	S
ClosedWorkload	S	S	S	S
EntryLevelSystemCall	S	S	S	S
ExternalCallAction	S	S	S	S
InternalAction	S[(a)]	S	S	S
Usage Model Branch	S	S	S	S
Probabilistic BranchAction	S	S	S	S
Guarded BranchAction	S	S	S	S
Usage Model Loop	S	S	S	S
LoopAction	S	S	S	S
CollectionIteratorAction	S	C	S	S
ForkAction	C	C	S	S
Acquire-/ReleaseAction	N	N	S	S
SystemSpecifiedExecutionTime	C	C	S	S
ComponentSpecifiedExecutionTime	N	C	S	C[(b)]
CompositeComponent	S	S	S	S
SubSystem	S	S	S	S
LinkingResource	N	N	S	N[(c)]
#Supported	13	12	18	16
#Not supported	3	2	0	1
#Crashed	2	4	0	1

Table 7.1: Feature Support Evaluation Results

Three table entries need additional explanation:

(a) The LQNS solver crashed in the scenario put together to test the *InternalAction* feature. As *InternalActions* are used in most of the other scenarios as well, and LQNS had no problems with that, we assume that the crash must be due to some other special characteristic of that particular scenario.

(b) The PCM technical reference describes the semantics of a *ComponentSpecifiedExecutionTime*. It is likely that the SimuCom simulator will support the feature in future versions.

(c) The launch configuration user interface for SimuCom contains a checkbox to enable simulation of *LinkingResources*. As enabling that feature caused a crash, it was deactivated for the evaluation. It is reasonable to assume that this feature is only broken in the version of SimuCom used for this thesis. Future versions are likely to support *LinkingResources* again.

Overall the SimQPN-based solver developed in this thesis has a very satisfactory feature support. A much broader range of PCM features is covered compared to the existing solvers based on the transformation to LQNs. The feature scenarios are important to validate the mapping. This is especially important for features not used in any of the case studies described in Chapter 8. The case studies examine the more commonly used features in much more realistic contexts and further increase the confidence in the results.

The LQNS and LQSim solvers are less reliable and cannot be recommended from the point of view of feature support. A superior results precision and runtime can still justify their use. This is examined in the case studies as well.

8. Case Studies

This section describes the case studies, the second part of the evaluation, in more detail. The derivation of metrics, assumptions and evaluation environment have already been covered in Chapter 7. Section 8.1 provides an overview and presents the schema after which a number of scenarios from different sources are evaluated and presented. Section 8.2 explains in detail the steps that were taken to gather the case study data. One section follows for each scenario (Section 8.3, Section 8.4, Section 8.5, Section 8.6, Section 8.7). The chapter concludes with an overall summary of the case study based on the results of the individual scenarios.

8.1 Overview

As mentioned in Section 7.1, the case studies are conducted to gain insight into the behavior of each of the evaluated solvers when confronted with large PCM instances of realistic complexity. Both the results precision and the analysis overhead are evaluated. Their relationship allows to estimate the precision that can be achieved in a given amount of time and helps in determining under which circumstances each solver should be used.

As there is great variance between realistic models, a single model is insufficient to reach a sound conclusion. Therefore, several models from different sources have been selected. In case different usage scenarios are available, the varied PCM instances are analyzed as well. Each case study model is evaluated according to the following schema:

1. Present an overview of the model and the PCM instances and variations to be analyzed.
2. Describe in more detail the complexity and characteristics of the PCM instances. The discussion is based on size metrics like the number of *AssemblyContext*s of the input PCM instance.
3. Present the evaluation results and the conclusions which can be drawn from the results.

The result data is presented uniformly throughout this chapter. The processing resource utilization metrics are represented as $U_{ResourceContainer\_ProcessingResourceType}$. $R_{ScenarioName}$ stands for the response time of a scenario, $X_{ScenarioName}$ for the throughput. The runtime in seconds is represented with $T_{Analysis}$. Each combination of PCM instance and logical simulated runtime has its own table. In several cases, the workload for that PCM instance is varied as well. The tables show the metric name in the first column. The second column presents the mean value $\bar{x}$of the SimuCom reference measurements. To achieve sufficient statistical significance, each experiment is run 30 times. A group of three additional columns is added for each of the evaluated solvers (SimQPN, LQNS and LQSim). The mean value $\bar{x}$is presented in the first column of the group. The second column contains the relative difference relDiffcompared to the SimuCom reference solver. The relative difference is $\frac{mean_{solver} - mean_{reference}}{mean_{reference}}$. For the response time, utilization and throughput metrics, relDiffrepresents the relative error of the measurements. For the execution time, it represents the reduction or addition in execution time compared to the reference solver. The third column (statSig) of the group shows whether or not the difference between the solver mean and the reference mean is statistically significant at a 95% confidence level. This is evaluated using an unpaired Student's t-test. If not enough data is available for the t-test, relDiffis shown, but statSigis omitted. If the t-test succeeds, but the difference is not statistically significant, relDiffis omitted. If a PCM instance causes a solver to crash, the data is omitted as well (showing a "-").

It is important to note that both LQSim and especially LQNS have a multitude of configuration options. Unfortunately, there was not enough time in this thesis to tune those solvers to achieve the best results. For LQSim, the same logical simulation time as for the other simulation based solvers was used. For LQNS the standard settings for a 95% convergence were used. The results for those solvers should therefore not be considered the best results possible. They are still useful to clarify situations in which the SimQPN solver and SimuCom solver have significantly different results. The results also provide a rough idea about how exact the solvers are and how fast they run.

One issue that turned up during the evaluation is that the SimQPN simulator execution time is larger the first time a simulation is executed compared to following simulation runs. In cases where this value stood out, a 31^{st} run was conducted, replacing the results of the first run. It is therefore assumed that executing a number of simulations in a row is the norm rather than an exception. The exact source of the effect is unclear but it is possible that some parts of SimQPN are kept as static fields, which have less memory management overhead the second time they are used.

In addition to the presented quantitative evaluation, a qualitative evaluation of the *UsageScenario* response time distributions between SimuCom and SimQPN is conducted. As the measurement with SimQPN involves an experimental feature (see Section 5.3.4), these results should only be seen as a general indication of what is possible. The histograms use close to 30 buckets, equally dividing the response time between 0 seconds and the maximum value encountered. The dimension and scaling of both axis are the same for each pair of histograms.

8.2 Steps

To gather the data for each of the case studies, the following steps were executed:

1. Identify the different model configurations in the originating project and create separate model instances for each.
2. Run the SimuCom solver and check if the results given in the project can be reproduced.
3. If no results are known or if the results cannot be reproduced, choose one configuration and modify it until it is solvable. Reduce the workload in case any of the processing resources are overloaded.
4. Compute the relative error of the mean of each solver compared to the SimuCom reference solver. To determine, if that error is statistically significant, a confidence

interval for the difference of means between the SimuCom result and each other solver is computed. The computations are carried out for each of the following metrics: mean response time and mean throughput for each *UsageScenario*, mean utilization for each *ProcessingResourceSpecification* (excluding DELAY resources) and (real) execution time. The current implementation of the SimQPN solver can only be run for a fixed amount of logical simulated seconds. It is assumed that the examined system reaches a steady state and therefore a higher runtime leads to better results. The time at which the steady state is reached depends on the model. The following substeps are executed to determine appropriate fixed simulation times for each model configuration and to compute the results.

a) Run the SimuCom solver on each of the resulting configurations using a relative precision termination condition of 95% for a 20% relative error (10% half width) confidence interval for the mean. The most important *UsageScenario* should be chosen as the target of this criterion. If the simulation converges, note the logical simulation finish time. If the the simulation run does not converge, choose the logical time it takes to simulate 100000 measurements instead. Once a number of logical run times have succesfully been identified, choose the largest runtime T as the basis for the following evaluation runs.

b) Configure each simulation-based solver to run a fixed amount of $(1/2)T$, T, and $2T$ and simulate each configuration 30 times. Configure LQNS to use a 95% confidence level and run it once (as it is deterministic and produces the same result every time).

5. Evaluate the results.

6. Choose a workload that puts significant load on the system, but does not overload it. Compute the response time distribution of each *UsageScenario* and produce histograms that use the same dimension and bucket sizes for the best comparison possible. This step is only available for SimuCom and SimQPN. The LQNS and LQSim solvers only support mean response times. With SimQPN the response time distribution is measured using the probes feature (see Section 5.3.4). This feature is still experimental and the results should be interpreted accordingly. The response time distributions are not evaluated quantitatively. Only a single experiment run is conducted and it is evaluated, how useful the histograms are to a performance analyst to gain additional insights.

8.3 Case Study 1: SPECjAppServer2004_Next

8.3.1 Overview

The SPECjAppServer2004_Next model is taken from [BKK09]. SPECjAppServer2004 is a benchmark developed by the Standard Performance Evaluation Corporation (SPEC). SPECjAppServer2004_Next is a beta version of the successor to the SPECjAppServer2004 benchmark. It is a Java EE benchmark developed by SPEC's Java subcommittee for measuring the performance and scalability of Java EE-based application servers. The benchmark aims at reproducing the behavior of a typical Java EE-based application. Therefore, it can be considered a system of realistic size and complexity.

In [BKK09] a PCM instance of the SPECjAppServer2004_Next benchmark was used to evaluate a method for automated extraction of Palladio Component Models from running enterprise java applications. To a large degree, the PCM instance was created by the extraction algorithm. Three different usage scenarios are considered in the paper (scenario 1, 2a and 2b). For this case study the models named "Model A" for the scenarios 1 and 2a were available. The scenarios will now be referred to as scenario A and scenario B. Each scenario is evaluated using a number of different workloads.

In PCM terms, this is a single PCM system model with a number of usage models each containing a single scenario and workload combination. The following variations were analyzed for this case study:

- Scenario A, throughput 13.315 requests/s
- Scenario A, throughput 33.627 requests/s
- Scenario A, throughput 49.925 requests/s
- Scenario A, throughput 71.120 requests/s
- Scenario B, throughput 11.254 requests/s
- Scenario B, throughput 22.211 requests/s
- Scenario B, throughput 33.898 requests/s
- Scenario B, throughput 43.691 requests/s

As the scenarios are very similar regarding their model structure, a single logical reference time was determined for all scenarios. The highest workload for each scenario was discarded as the system was saturated and the relative stopping criterion of PCM did not converge. Most logical runs finished in between 600 and 1200 simulated seconds. However, the scenario A/11.254 throughput combination took 3200 simulated seconds once. To be safe, 4000 simulated seconds were chosen as the base time T.

8.3.2 Complexity

The two evaluated scenarios differ only in their usage model and workload. Both scenarios have only one *UsageScenario*. Scenario A has a very simple usage model with one *EntryLevelSystemCall*, no loops and no branches. The workload is open and is varied between 13 and 71 requests/s. Scenario B has a more complex usage model with 6 *EntryLevelSystemCall*s and one loop. The workload is varied between 11 and 43 requests/s.

The system part is the same for both scenarios. 7 *AssemblyContext*s reference 7 different *BasicComponent*s. No *CompositeComponent*s or *SubSystem*s are referenced. The referenced SEFFs contain 14 *ExternalCallAction*s, one loop and 4 branches. One *PassiveResource* is referenced.

Regarding stochastic expressions, the Exp-function is used in the workload as the inter-arrival time. IntPMF is used in the SEFFs.

8.3.3 Results

For this case study, two sets of results were obtained. The initial set of results showed an accuracy which was out of line compared to the other case study results. The issue was traced to a temporary simplification in the SimQPN solver implementation. As the SimQPN simulator support for empirical distributions does not meet the requirements of the PCM-to-QPN mapping, mean values and a deterministic distribution are used instead. As this case study is the only case study that uses exponentially distributed inter arrival times in the workload specification (using the Exp-function), the error introduced by that simplification does not show in the other case studies. For scenario A, a second set of results was obtained in which the transformation was manually modified to insert the correct distribution, showing much better results. This shortcoming of the SimQPN simulator and solver will be resolved in future versions.

Throughput: 13.315 req/s										
	SimuCom	**SimQPN**			**LQNS**			**LQSIM**		
Metric	$\bar{x}$	$\bar{x}$	relDiff	statSig	$\bar{x}$	relDiff	statSig	$\bar{x}$	relDiff	statSig
U_{WLS_CPU}	0.186	0.196	5.31%	yes	-	-	-	-	-	-
U_{DBS_CPU}	0.0555	0.058	4.51%	yes	-	-	-	-	-	-
$R_{Scenario}$	0.0215	0.018	-16.4%	yes	-	-	-	-	-	-
$T_{Analysis}$	41.1	3.31	-92%	yes	-	-	-	-	-	-

Throughput: 33.627 req/s										
	SimuCom	**SimQPN**			**LQNS**			**LQSIM**		
Metric	$\bar{x}$	$\bar{x}$	relDiff	statSig	$\bar{x}$	relDiff	statSig	$\bar{x}$	relDiff	statSig
U_{WLS_CPU}	0.469	0.493	4.98%	yes	-	-	-	-	-	-
U_{DBS_CPU}	0.140	0.147	4.77%	yes	-	-	-	-	-	-
$R_{Scenario}$	0.0312	0.02	-35.8%	yes	-	-	-	-	-	-
$T_{Analysis}$	100	5.44	-94.6%	yes	-	-	-	-	-	-

Throughput: 49.925 req/s										
	SimuCom	**SimQPN**			**LQNS**			**LQSIM**		
Metric	$\bar{x}$	$\bar{x}$	relDiff	statSig	$\bar{x}$	relDiff	statSig	$\bar{x}$	relDiff	statSig
U_{WLS_CPU}	0.697	0.732	5.04%	yes	-	-	-	-	-	-
U_{DBS_CPU}	0.208	0.219	5.17%	yes	-	-	-	-	-	-
$R_{Scenario}$	0.051	0.0281	-44.9%	yes	-	-	-	-	-	-
$T_{Analysis}$	141	7.31	-94.8%	yes	-	-	-	-	-	-

Throughput: 71.120 req/s										
	SimuCom	**SimQPN**			**LQNS**			**LQSIM**		
Metric	$\bar{x}$	$\bar{x}$	relDiff	statSig	$\bar{x}$	relDiff	statSig	$\bar{x}$	relDiff	statSig
U_{WLS_CPU}	0.989	1.0	1.15%	yes	-	-	-	-	-	-
U_{DBS_CPU}	0.292	0.298	2.06%	yes	-	-	-	-	-	-
$R_{Scenario}$	1.06	45.9	4228%$^{(a)}$	yes	-	-	-	-	-	-
$T_{Analysis}$	41.8	310	641%$^{(a)}$	yes	-	-	-	-	-	-

Table 8.1: SPECjAppServer2004_Next Scenario A Evaluation Results for 2000 sec

Table 8.1, Table 8.2 and Table 8.3 show the initial results for scenario A for the three logical simulation runtimes (2000, 4000 and 8000 seconds). No data is available for LQNS and LQSim because they could not handle the Exp-function used in the workload specification. First of all, the values marked with (a) deserve special attention: For the 71.120 requests/s workloads the values for the scenario mean response times are 40 to almost 200 times higher than the values of SimuCom. The reason is that for such a high workload one of the CPUs is at maximum capacity and the scenario uses an open workload. SimuCom tries to allocate a new thread for each new request. As the requests are spawned faster than they can be serviced, an internal out of memory exception is thrown and SimuCom aborts the simulation prematurely. SimQPN runs the full specified duration as it does not run out of memory. The system never reaches a stable state. The numbers for the mean response times have therefore no meaning as they rise without bound with the simulation time. The numbers have only been included here as a guide how such numbers are to be interpreted by a performance analyst. The important information is the bottleneck resource, which has correctly been identified by both solvers.

The other workloads show that the processing resource utilization prediction of SimQPN is always about 5% higher than the prediction by SimuCom. The explanation for this is that the DependencySolver reduces the stochastic expression for the inter-arrival time of the workload to a generic distribution. In this case, the computation shows an imprecision resulting in a mean value about 5% higher than the original mean of the specified Exp-Function.

In this initial results set the values for the mean response times show a significant error. For a low workload, the estimate is about 17% lower than the SimuCom estimate. The error increases with the workload, reaching almost 45% for a high workload. Table 8.4 shows the corrected set of results for scenario A. Apart from the type of distribution used, the mean value of the inter-arrival time was also corrected. The results now show an excellent prediction with an error below 0.2% for the utilizations and below 1.5% for the mean response times. This shows that the error was only caused by a temporary implementation issue and that it is not inherent in the PCM-to-QPN mapping.

Table 8.5, Table 8.6 and Table 8.7 show the results for scenario B. This time, the highest workload does not overload one of the CPUs and the analysis reaches steady state. Again we notice the 5% error in the CPU utilization estimates. The scenario mean response times, while still lower than the SimuCom estimates, however, they do not exceed an 8% error this time. At a first sight this is surprising. It can, however, easily be explained with the loop which is introduced in the usage model of scenario B. The loop calls a delay resource three time with a demand of 333ms each. Each request therefore runs at least a fixed amount of 999ms independently from all other circumstances. As the response times are on average very close to 1000ms, we naturally reach a much smaller error. If the relative error is computed after substracting the fixed 999ms from the results, we see a situation comparable to scenario A. At 11.254 requests/s and 8000 seconds logical simulation time we have a corrected relative error of about 5%. At 43.691 requests/s and 8000 seconds we have an error of about 35%. The reason for the error is the same as in scenario A.

There are no notable differences in the results between the different logical runtimes.

Figure 8.1 shows the response time distributions for scenario A with a workload of 43.691 req/s and a logical simulation time of 8000 seconds of SimuCom and SimQPN. The SimQPN distribution shows a much lower peak, which matches the error in the mean response time predictions.

Figure 8.2 shows the diagram for the manually corrected set of experiments. The peaks now match. However, the distribution of SimQPN stretches further out to the right. This is caused by the approximation introduced in the loop mapping (see Section 5.2.5).

Throughput: 13.315 req/s										
	SimuCom	**SimQPN**			**LQNS**			**LQSIM**		
Metric	$\bar{x}$	$\bar{x}$	relDiff	statSig	$\bar{x}$	relDiff	statSig	$\bar{x}$	relDiff	statSig
U_{WLS_CPU}	0.185	0.195	5.34%	yes	-	-	-	-	-	-
U_{DBS_CPU}	0.0554	0.058	4.66%	yes	-	-	-	-	-	-
$R_{Scenario}$	0.0215	0.018	-16.4%	yes	-	-	-	-	-	-
$T_{Analysis}$	81.2	4.77	-94.1%	yes	-	-	-	-	-	-

Throughput: 33.627 req/s										
	SimuCom	**SimQPN**			**LQNS**			**LQSIM**		
Metric	$\bar{x}$	$\bar{x}$	relDiff	statSig	$\bar{x}$	relDiff	statSig	$\bar{x}$	relDiff	statSig
U_{WLS_CPU}	0.469	0.493	4.98%	yes	-	-	-	-	-	-
U_{DBS_CPU}	0.140	0.147	4.78%	yes	-	-	-	-	-	-
$R_{Scenario}$	0.0311	0.02	-35.8%	yes	-	-	-	-	-	-
$T_{Analysis}$	199	9	-95.5%	yes	-	-	-	-	-	-

Throughput: 49.925 req/s										
	SimuCom	**SimQPN**			**LQNS**			**LQSIM**		
Metric	$\bar{x}$	$\bar{x}$	relDiff	statSig	$\bar{x}$	relDiff	statSig	$\bar{x}$	relDiff	statSig
U_{WLS_CPU}	0.697	0.732	5.07%	yes	-	-	-	-	-	-
U_{DBS_CPU}	0.208	0.219	5.13%	yes	-	-	-	-	-	-
$R_{Scenario}$	0.0513	0.028	-45.4%	yes	-	-	-	-	-	-
$T_{Analysis}$	274	12.7	-95.4%	yes	-	-	-	-	-	-

Throughput: 71.120 req/s										
	SimuCom	**SimQPN**			**LQNS**			**LQSIM**		
Metric	$\bar{x}$	$\bar{x}$	relDiff	statSig	$\bar{x}$	relDiff	statSig	$\bar{x}$	relDiff	statSig
U_{WLS_CPU}	0.988	1.0	1.17%	yes	-	-	-	-	-	-
U_{DBS_CPU}	0.294	0.298	1.69%	yes	-	-	-	-	-	-
$R_{Scenario}$	0.953	90.7	9417%$^{(a)}$	yes	-	-	-	-	-	-
$T_{Analysis}$	78.6	1156	1370%$^{(a)}$	yes	-	-	-	-	-	-

Table 8.2: SPECjAppServer2004_Next Scenario A Evaluation Results for 4000 sec

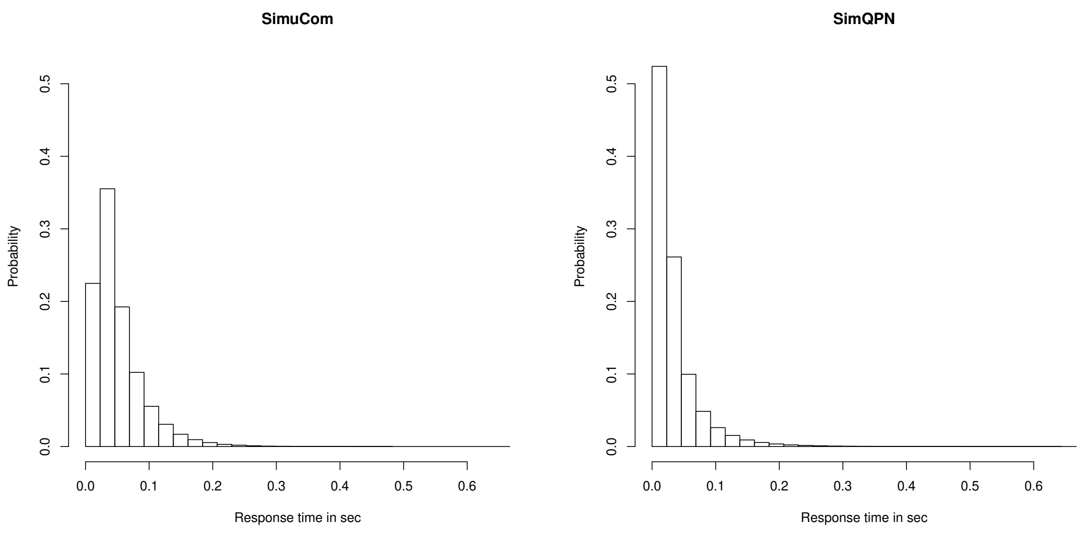


Figure 8.1: SPECjAppServer2004_Next SimuCom vs SimQPN Histograms ScenarioA 49.925 req/s 8000sec

Throughput: 13.315 req/s										
	SimuCom	**SimQPN**			**LQNS**			**LQSIM**		
Metric	$\bar{x}$	$\bar{x}$	relDiff	statSig	$\bar{x}$	relDiff	statSig	$\bar{x}$	relDiff	statSig
U_{WLS_CPU}	0.186	0.195	5.08%	yes	-	-	-	-	-	-
U_{DBS_CPU}	0.0556	0.058	4.41%	yes	-	-	-	-	-	-
$R_{Scenario}$	0.0216	0.018	-16.5%	yes	-	-	-	-	-	-
$T_{Analysis}$	176	7.35	-95.8%	yes	-	-	-	-	-	-

Throughput: 33.627 req/s										
	SimuCom	**SimQPN**			**LQNS**			**LQSIM**		
Metric	$\bar{x}$	$\bar{x}$	relDiff	statSig	$\bar{x}$	relDiff	statSig	$\bar{x}$	relDiff	statSig
U_{WLS_CPU}	0.469	0.493	5.14%	yes	-	-	-	-	-	-
U_{DBS_CPU}	0.140	0.147	4.83%	yes	-	-	-	-	-	-
$R_{Scenario}$	0.0311	0.02	-35.7%	yes	-	-	-	-	-	-
$T_{Analysis}$	454	15.7	-96.5%	yes	-	-	-	-	-	-

Throughput: 49.925 req/s										
	SimuCom	**SimQPN**			**LQNS**			**LQSIM**		
Metric	$\bar{x}$	$\bar{x}$	relDiff	statSig	$\bar{x}$	relDiff	statSig	$\bar{x}$	relDiff	statSig
U_{WLS_CPU}	0.696	0.732	5.14%	yes	-	-	-	-	-	-
U_{DBS_CPU}	0.208	0.219	5.23%	yes	-	-	-	-	-	-
$R_{Scenario}$	0.0512	0.028	-45.3%	yes	-	-	-	-	-	-
$T_{Analysis}$	557	23	-95.9%	yes	-	-	-	-	-	-

Throughput: 71.120 req/s										
	SimuCom	**SimQPN**			**LQNS**			**LQSIM**		
Metric	$\bar{x}$	$\bar{x}$	relDiff	statSig	$\bar{x}$	relDiff	statSig	$\bar{x}$	relDiff	statSig
U_{WLS_CPU}	0.988	1.0	1.19%	-	-	-	-	-	-	-
U_{DBS_CPU}	0.293	0.298	1.61%	-	-	-	-	-	-	-
$R_{Scenario}$	0.934	184	19609%(a)	-	-	-	-	-	-	-
$T_{Analysis}$	38.7	4828	12387%(a)	-	-	-	-	-	-	-

Table 8.3: SPECjAppServer2004_Next Scenario A Evaluation Results for 8000 sec

Throughput: 13.315 req/s										
	SimuCom	**SimQPN**			**LQNS**			**LQSIM**		
Metric	$\bar{x}$	$\bar{x}$	relDiff	statSig	$\bar{x}$	relDiff	statSig	$\bar{x}$	relDiff	statSig
U_{WLS_CPU}	0.186	0.186	-	no	-	-	-	-	-	-
U_{DBS_CPU}	0.0556	0.0555	-	no	-	-	-	-	-	-
$R_{Scenario}$	0.0216	0.0219	1.46%	yes	-	-	-	-	-	-
$T_{Analysis}$	176	7.86	-95.5%	yes	-	-	-	-	-	-

Throughput: 33.627 req/s										
	SimuCom	**SimQPN**			**LQNS**			**LQSIM**		
Metric	$\bar{x}$	$\bar{x}$	relDiff	statSig	$\bar{x}$	relDiff	statSig	$\bar{x}$	relDiff	statSig
U_{WLS_CPU}	0.469	0.469	-	no	-	-	-	-	-	-
U_{DBS_CPU}	0.140	0.140	-	no	-	-	-	-	-	-
$R_{Scenario}$	0.0311	0.031	-0.405%	yes	-	-	-	-	-	-
$T_{Analysis}$	454	17.3	-96.2%	yes	-	-	-	-	-	-

Throughput: 49.925 req/s										
	SimuCom	**SimQPN**			**LQNS**			**LQSIM**		
Metric	$\bar{x}$	$\bar{x}$	relDiff	statSig	$\bar{x}$	relDiff	statSig	$\bar{x}$	relDiff	statSig
U_{WLS_CPU}	0.696	0.697	0.153%	yes	-	-	-	-	-	-
U_{DBS_CPU}	0.208	0.208	-	no	-	-	-	-	-	-
$R_{Scenario}$	0.0512	0.0511	-	no	-	-	-	-	-	-
$T_{Analysis}$	557	24.7	-95.6%	yes	-	-	-	-	-	-

Table 8.4: SPECjAppServer2004_Next Scenario A Evaluation Results for 8000 sec (corrected)

Throughput: 11.254 req/s										
	SimuCom	**SimQPN**			**LQNS**			**LQSIM**		
Metric	$\bar{x}$	$\bar{x}$	relDiff	statSig	$\bar{x}$	relDiff	statSig	$\bar{x}$	relDiff	statSig
U_{WLS_CPU}	0.235	0.247	5.16%	yes	-	-	-	-	-	-
U_{DBS_CPU}	0.133	0.14	5.1%	yes	-	-	-	-	-	-
$R_{Scenario}$	1.04	1.04	0.161%	yes	-	-	-	-	-	-
$T_{Analysis}$	4.85	65.5	-92.6%	yes	-	-	-	-	-	-

Throughput: 22.211 req/s										
	SimuCom	**SimQPN**			**LQNS**			**LQSIM**		
Metric	$\bar{x}$	$\bar{x}$	relDiff	statSig	$\bar{x}$	relDiff	statSig	$\bar{x}$	relDiff	statSig
U_{WLS_CPU}	0.463	0.487	5.21%	yes	-	-	-	-	-	-
U_{DBS_CPU}	0.263	0.276	5.18%	yes	-	-	-	-	-	-
$R_{Scenario}$	1.05	1.05	-0.731%	yes	-	-	-	-	-	-
$T_{Analysis}$	126	7.37	-94.2%	yes	-	-	-	-	-	-

Throughput: 33.898 req/s										
	SimuCom	**SimQPN**			**LQNS**			**LQSIM**		
Metric	$\bar{x}$	$\bar{x}$	relDiff	statSig	$\bar{x}$	relDiff	statSig	$\bar{x}$	relDiff	statSig
U_{WLS_CPU}	0.708	0.744	5.09%	yes	-	-	-	-	-	-
U_{DBS_CPU}	0.401	0.422	5.13%	yes	-	-	-	-	-	-
$R_{Scenario}$	1.09	1.06	-2.93%	yes	-	-	-	-	-	-
$T_{Analysis}$	186	10.4	-94.4%	yes	-	-	-	-	-	-

Throughput: 43.691 req/s										
	SimuCom	**SimQPN**			**LQNS**			**LQSIM**		
Metric	$\bar{x}$	$\bar{x}$	relDiff	statSig	$\bar{x}$	relDiff	statSig	$\bar{x}$	relDiff	statSig
U_{WLS_CPU}	0.912	0.958	5.08%	yes	-	-	-	-	-	-
U_{DBS_CPU}	0.517	0.544	5.05%	yes	-	-	-	-	-	-
$R_{Scenario}$	1.26	1.17	-7.37%	yes	-	-	-	-	-	-
$T_{Analysis}$	228	14	-93.9%	yes	-	-	-	-	-	-

Table 8.5: SPECjAppServer2004_Next Scenario B Evaluation Results for 2000 sec

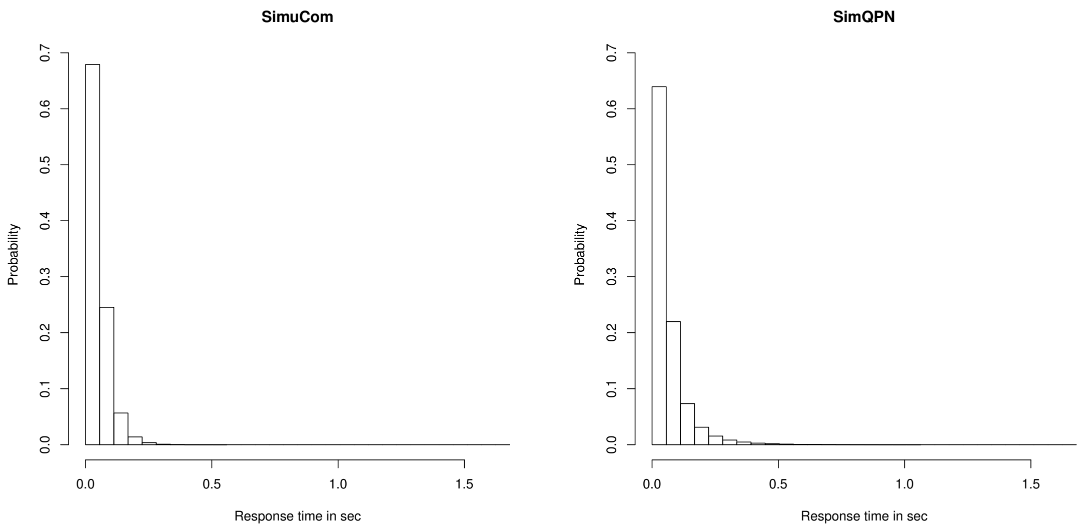


Figure 8.2: SPECjAppServer2004_Next SimuCom vs SimQPN Histograms (corrected) ScenarioA 49.925 req/s 8000sec

Throughput: 11.254 req/s										
	SimuCom	**SimQPN**			**LQNS**			**LQSIM**		
Metric	$\bar{x}$	$\bar{x}$	relDiff	statSig	$\bar{x}$	relDiff	statSig	$\bar{x}$	relDiff	statSig
U_{WLS_CPU}	0.235	0.247	5.16%	yes	-	-	-	-	-	-
U_{DBS_CPU}	0.133	0.14	5.13%	yes	-	-	-	-	-	-
$R_{Scenario}$	1.04	1.04	0.226%	yes	-	-	-	-	-	-
$T_{Analysis}$	128	7.3	-94.3%	yes	-	-	-	-	-	-

Throughput: 22.211 req/s										
	SimuCom	**SimQPN**			**LQNS**			**LQSIM**		
Metric	$\bar{x}$	$\bar{x}$	relDiff	statSig	$\bar{x}$	relDiff	statSig	$\bar{x}$	relDiff	statSig
U_{WLS_CPU}	0.464	0.487	5.11%	yes	-	-	-	-	-	-
U_{DBS_CPU}	0.263	0.276	5.05%	yes	-	-	-	-	-	-
$R_{Scenario}$	1.05	1.05	-0.792%	yes	-	-	-	-	-	-
$T_{Analysis}$	250	12.4	-95%	yes	-	-	-	-	-	-

Throughput: 33.898 req/s										
	SimuCom	**SimQPN**			**LQNS**			**LQSIM**		
Metric	$\bar{x}$	$\bar{x}$	relDiff	statSig	$\bar{x}$	relDiff	statSig	$\bar{x}$	relDiff	statSig
U_{WLS_CPU}	0.708	0.743	5.03%	yes	-	-	-	-	-	-
U_{DBS_CPU}	0.402	0.422	5.1%	yes	-	-	-	-	-	-
$R_{Scenario}$	1.09	1.06	-2.98%	yes	-	-	-	-	-	-
$T_{Analysis}$	370	18.4	-95%	yes	-	-	-	-	-	-

Throughput: 43.691 req/s										
	SimuCom	**SimQPN**			**LQNS**			**LQSIM**		
Metric	$\bar{x}$	$\bar{x}$	relDiff	statSig	$\bar{x}$	relDiff	statSig	$\bar{x}$	relDiff	statSig
U_{WLS_CPU}	0.912	0.958	5.09%	yes	-	-	-	-	-	-
U_{DBS_CPU}	0.517	0.544	5.12%	yes	-	-	-	-	-	-
$R_{Scenario}$	1.26	1.17	-7.37%	yes	-	-	-	-	-	-
$T_{Analysis}$	457	25.8	-94.4%	yes	-	-	-	-	-	-

Table 8.6: SPECjAppServer2004_Next Scenario B Evaluation Results for 4000 sec

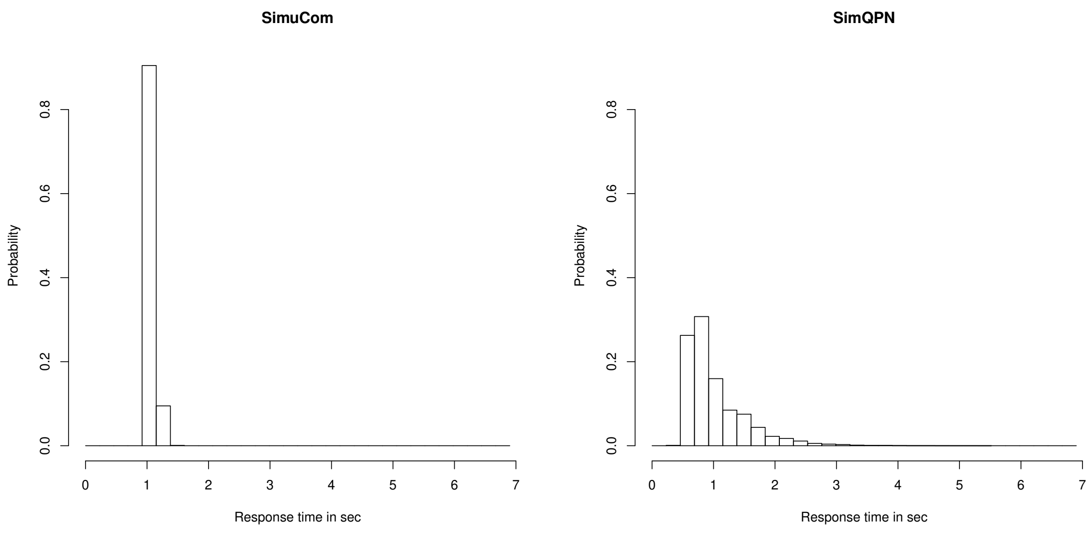


Figure 8.3: SPECjAppServer2004_Next SimuCom vs SimQPN Histograms ScenarioB 33.898 req/s 8000sec

Throughput: 11.254 req/s										
	SimuCom	**SimQPN**			**LQNS**			**LQSIM**		
Metric	$\bar{x}$	$\bar{x}$	relDiff	statSig	$\bar{x}$	relDiff	statSig	$\bar{x}$	relDiff	statSig
U_{WLS_CPU}	0.235	0.247	5%	yes	-	-	-	-	-	-
U_{DBS_CPU}	0.133	0.14	4.95%	yes	-	-	-	-	-	-
$R_{Scenario}$	1.04	1.04	0.207%	yes	-	-	-	-	-	-
$T_{Analysis}$	253	12.5	-95%	yes	-	-	-	-	-	-

Throughput: 22.211 req/s										
	SimuCom	**SimQPN**			**LQNS**			**LQSIM**		
Metric	$\bar{x}$	$\bar{x}$	relDiff	statSig	$\bar{x}$	relDiff	statSig	$\bar{x}$	relDiff	statSig
U_{WLS_CPU}	0.464	0.487	4.99%	yes	-	-	-	-	-	-
U_{DBS_CPU}	0.263	0.276	4.99%	yes	-	-	-	-	-	-
$R_{Scenario}$	1.05	1.05	-0.78%	yes	-	-	-	-	-	-
$T_{Analysis}$	510	23.1	-95.5%	yes	-	-	-	-	-	-

Throughput: 33.898 req/s										
	SimuCom	**SimQPN**			**LQNS**			**LQSIM**		
Metric	$\bar{x}$	$\bar{x}$	relDiff	statSig	$\bar{x}$	relDiff	statSig	$\bar{x}$	relDiff	statSig
U_{WLS_CPU}	0.708	0.744	5.05%	yes	-	-	-	-	-	-
U_{DBS_CPU}	0.402	0.422	5.06%	yes	-	-	-	-	-	-
$R_{Scenario}$	1.09	1.06	-2.99%	yes	-	-	-	-	-	-
$T_{Analysis}$	777	35.2	-95.5%	yes	-	-	-	-	-	-

Throughput: 43.691 req/s										
	SimuCom	**SimQPN**			**LQNS**			**LQSIM**		
Metric	$\bar{x}$	$\bar{x}$	relDiff	statSig	$\bar{x}$	relDiff	statSig	$\bar{x}$	relDiff	statSig
U_{WLS_CPU}	0.912	0.959	5.14%	yes	-	-	-	-	-	-
U_{DBS_CPU}	0.517	0.544	5.13%	yes	-	-	-	-	-	-
$R_{Scenario}$	1.26	1.17	-7.32%	yes	-	-	-	-	-	-
$T_{Analysis}$	926	50.6	-94.5%	yes	-	-	-	-	-	-

Table 8.7: SPECjAppServer2004_Next Scenario B Evaluation Results for 8000 sec

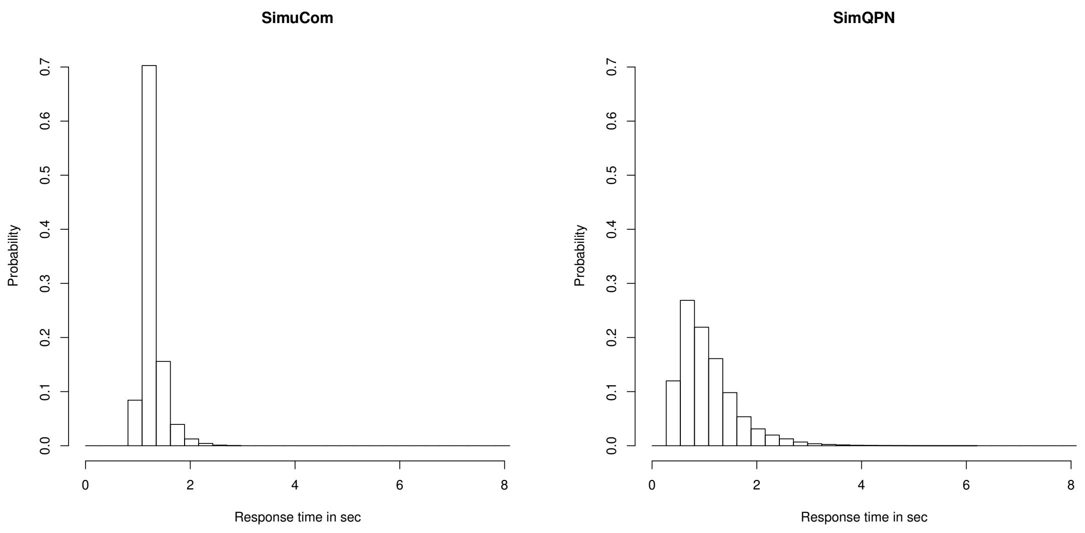


Figure 8.4: SPECjAppServer2004_Next SimuCom vs SimQPN Histograms ScenarioB 43.691 req/s 8000sec

Looking at Figure 8.3, the distributions for scenario B with 33.898 req/s, and Figure 8.4, with 43.691 req/s, the match is clearly worse. The SimuCom results are almost constant, while the SimQPN results distribution looks roughly like a Poisson distribution. In this case the approximation introduced in the loop mapping (see Section 5.2.5) cause a noticeable error in the distribution. The mapping allows for cases in which the usage model loop is executed only once, and for cases in which the loop is executed more than the specified three times. With over 300ms delay in the loop body of the usage scenario, even a reduction of one iteration has a noticeable effect.

Looking at the execution times of the SimQPN run with probes, the runs took about 1.8 times as long as without the probes measurements. However, the execution time is still about an order of magnitude lower than the execution time of SimuCom.

8.4 Case Study 2: ABB Demonstrator

8.4.1 Overview

The PCM instance of the ABB demonstrator model was made available in the context of an internship at ABB Research in Ladenburg, Germany. It was created to evaluate the Q-ImPrESS method of the Q-ImPrESS joint research project. The system used as the ABB demonstrator for the Q-ImPrESS method is a large distributed [process] control system by ABB [qim, D7.1]. We therefore assume that it meets our requirements of being a representative system of a real industrial application.

In this case study, the workloads for the two main usage scenarios ("Retrieve Data" and "History Retrieve") are varied. In the following, they are referred to as scenario A and scenario B. To limit the number of scenario/workload combinations, the workloads for the two scenarios are varied independently. With the scenario B throughput fixed at 5 requests/s, we analyzed:

- Scenario A throughput 30 requests/s
- Scenario A throughput 60 requests/s
- Scenario A throughput 90 requests/s
- Scenario A throughput 120 requests/s
- Scenario A throughput 150 requests/s

Keeping the scenario A throughput at 30 requests/s, we analyzed:

- Scenario B throughput 3 requests/s
- Scenario B throughput 4 requests/s
- Scenario B throughput 5 requests/s
- Scenario B throughput 10 requests/s

As all variations are based on the same PCM system composition, a single logical reference simulation time has been derived. A number of test runs showed that the PCM relative stopping criterion for the main scenario (scenario A) did not appear to converge. Therefore the time until 100000 requests were finished, 3000 logical seconds, was taken as the base time T.

8.4.2 Complexity

The ABB Demonstrator PCM instance is constructed following a simple pattern. It consists of a number of *BasicComponent*s, offering a single service each. Each SEFF consists of a single *InternalAction*, followed by a branch with *ExternalCallAction*s to the other components of the system. The resource demands for the internal actions were derived from black-box utilization measurements of the individual components using the Service Demand Law. The branch probabilities were derived from inter-component request logs.

The usage model consists of four parallel *UsageScenario*s. Each *UsageScenario* is very simple and contains only one *EntryLevelSystemCall* and no loops or branches. All workloads are open workloads. They are varied between 30 and 120 req/s for the main scenario and between 3 and 10 req/s for the second most important scenario. The remaining scenarios run at 1 req/s and do not cause significant load on the system.

The system part of the model is of about average size with 10 referenced *AssemblyContext*s in 9 referenced *BasicComponent*s. No *CompositeComponent*s or *SubSystem*s are referenced. The referenced SEFFs contain 17 *ExternalCallAction*s. As mentioned before, there are 9 branches, one per component, and no loops. No *PassiveResource*s are referenced.

Apart from basic number arithmetic, no advanced stochastic expression constructs are used. Requests arrive at a deterministic and constant rate.

8.4.3 Results

Table 8.8, Table 8.9 and Table 8.10 show the results for the variation of the scenario A workload for each of the three logical simulation times (1500, 3000 and 6000 seconds). The processing resource utilizations are predicted very accurately by all solvers. The results of the SimQPN, LQNS and LQSim solvers mostly have no statistically significant difference to the SimuCom result. The remaining cases stay below a derivation of 0.5%.

In the prediction of the scenario mean response time the situation is more diverse. The workload appears to have no strong effect on the accuracy of prediction of the SimQPN solver. The results stay below a 10% error for scenario A and below a 2% error for scenario B, which are good values for a response time prediction. The two LQN-based solvers show no notable differences between each other. However, the accuracy of the prediction results for the mean response time of scenario A appears to decrease exponentially with an increasing workload for scenario A. The accuracy of the results for scenario B appears to decrease in a linear fashion. The prediction results for the lowest workload for scenario A (30 req/s) stay around a 5% error. For the highest workload for scenario A (150 req/s) the error goes up to over 70% for scenario A and to over 17% for scenario B.

Table 8.11, Table 8.12 and Table 8.13 show the results for the variation of the scenario B workload for the different runtimes. Compared to the scenario A results, there are no notable differences in the accuracy of the processing resource utilization predictions and in the SimQPN mean response time predictions. The LQN-based solvers show better predictions for the response times but the accuracy appears to decrease exponentially (this time for both scenarios) with an increasing workload. At 3 req/s for scenario B, the prediction results are very accurate with an error below 2%. At 10 req/s for scenario B, the error goes up to over 9% for scenario A and over 28% for scenario B.

There are no notable differences in the results between the different logical runtimes.

Figure 8.5 shows the response time distributions for scenario A with a workload of 120 req/s for scenario A, 5 req/s for scenario B, and a logical simulation time of 6000 seconds.

Throughputs: ScenarioA 30 req/s ScenarioB 5 req/s										
	SimuCom	**SimQPN**			**LQNS**			**LQSIM**		
Metric	$\bar{x}$	$\bar{x}$	relDiff	statSig	$\bar{x}$	relDiff	statSig	$\bar{x}$	relDiff	statSig
U_{AS_CPU}	0.123	0.123	0.196%	yes	0.123	-	no	0.122	-	no
U_{AS_CPU}	0.168	0.168	-	no	0.168	-	no	0.167	-	no
$R_{ScenarioA}$	0.00452	0.00477	5.42%	yes	0.00431	-4.67%	yes	0.00437	-3.37%	yes
$R_{ScenarioB}$	0.0383	0.038	-0.87%	yes	0.0409	6.65%	yes	0.0408	6.56%	yes
$T_{Analysis}$	18	6.83	-62%	yes	0.905	-95%	yes	1.48	-91.8%	yes

Throughputs: ScenarioA 60 req/s ScenarioB 5 req/s										
	SimuCom	**SimQPN**			**LQNS**			**LQSIM**		
Metric	$\bar{x}$	$\bar{x}$	relDiff	statSig	$\bar{x}$	relDiff	statSig	$\bar{x}$	relDiff	statSig
U_{AS_CPU}	0.219	0.219	-	no	0.219	-	no	0.219	-	no
U_{AS_CPU}	0.186	0.186	-	no	0.186	-	no	0.186	-	no
$R_{ScenarioA}$	0.00419	0.004	-4.64%	yes	0.00477	13.8%	yes	0.00488	16.4%	yes
$R_{ScenarioB}$	0.0386	0.039	1.05%	yes	0.0423	9.51%	yes	0.0422	9.26%	yes
$T_{Analysis}$	29.4	7.11	-75.8%	yes	0.749	-97.5%	yes	1.88	-93.6%	yes

Throughputs: ScenarioA 90 req/s ScenarioB 5 req/s										
	SimuCom	**SimQPN**			**LQNS**			**LQSIM**		
Metric	$\bar{x}$	$\bar{x}$	relDiff	statSig	$\bar{x}$	relDiff	statSig	$\bar{x}$	relDiff	statSig
U_{AS_CPU}	0.315	0.314	-	no	0.315	-	no	0.315	-	no
U_{AS_CPU}	0.204	0.204	-	no	0.204	-	no	0.205	-	no
$R_{ScenarioA}$	0.00410	0.004	-2.55%	yes	0.00539	31.4%	yes	0.00548	33.4%	yes
$R_{ScenarioB}$	0.0390	0.039	0.118%	yes	0.0439	12.6%	yes	0.0442	13.5%	yes
$T_{Analysis}$	41.8	7.4	-82.3%	yes	0.85	-98%	yes	2.39	-94.3%	yes

Throughputs: ScenarioA 120 req/s ScenarioB 5 req/s										
	SimuCom	**SimQPN**			**LQNS**			**LQSIM**		
Metric	$\bar{x}$	$\bar{x}$	relDiff	statSig	$\bar{x}$	relDiff	statSig	$\bar{x}$	relDiff	statSig
U_{AS_CPU}	0.411	0.411	-	no	0.411	-	no	0.410	-	no
U_{AS_CPU}	0.222	0.222	-	no	0.222	-	no	0.223	-	no
$R_{ScenarioA}$	0.00409	0.004	-2.18%	yes	0.00614	50.1%	yes	0.00629	53.8%	yes
$R_{ScenarioB}$	0.0394	0.0391	-0.783%	yes	0.0457	15.8%	yes	0.0458	16%	yes
$T_{Analysis}$	54	7.85	-85.5%	yes	0.902	-98.3%	yes	2.8	-94.8%	yes

Throughputs: ScenarioA 150 req/s ScenarioB 5 req/s										
	SimuCom	**SimQPN**			**LQNS**			**LQSIM**		
Metric	$\bar{x}$	$\bar{x}$	relDiff	statSig	$\bar{x}$	relDiff	statSig	$\bar{x}$	relDiff	statSig
U_{AS_CPU}	0.507	0.507	-	no	0.507	-	no	0.507	-	no
U_{AS_CPU}	0.24	0.24	-	no	0.24	-	no	0.240	-	no
$R_{ScenarioA}$	0.00418	0.004	-4.28%	yes	0.00715	71.2%	yes	0.00742	77.6%	yes
$R_{ScenarioB}$	0.0405	0.0407	0.492%	yes	0.0479	18.3%	yes	0.048	18.5%	yes
$T_{Analysis}$	66.8	8.22	-87.7%	yes	0.993	-98.5%	yes	3.36	-95%	yes

Table 8.8: ABB Demonstrator ScenarioA Variation Evaluation Results for 1500 sec

Throughputs: ScenarioA 30 req/s ScenarioB 5 req/s										
	SimuCom	**SimQPN**			**LQNS**			**LQSIM**		
Metric	$\bar{x}$	$\bar{x}$	relDiff	statSig	$\bar{x}$	relDiff	statSig	$\bar{x}$	relDiff	statSig
U_{AS_CPU}	0.123	0.123	0.241%	yes	0.123	-	no	0.122	-	no
U_{AS_CPU}	0.168	0.168	-	no	0.168	-	no	0.168	-	no
$R_{ScenarioA}$	0.00452	0.0049	8.32%	yes	0.00431	-4.71%	yes	0.00437	-3.34%	yes
$R_{ScenarioB}$	0.0383	0.038	-0.872%	yes	0.0409	6.65%	yes	0.0409	6.69%	yes
$T_{Analysis}$	32.4	7.15	-78%	yes	0.905	-97.2%	yes	1.99	-93.9%	yes

Throughputs: ScenarioA 60 req/s ScenarioB 5 req/s										
	SimuCom	**SimQPN**			**LQNS**			**LQSIM**		
Metric	$\bar{x}$	$\bar{x}$	relDiff	statSig	$\bar{x}$	relDiff	statSig	$\bar{x}$	relDiff	statSig
U_{AS_CPU}	0.219	0.219	0.120%	yes	0.219	-	no	0.218	-	no
U_{AS_CPU}	0.186	0.186	-	no	0.186	-	no	0.187	-	no
$R_{ScenarioA}$	0.00419	0.004	-4.5%	yes	0.00477	14.0%	yes	0.00486	16.0%	yes
$R_{ScenarioB}$	0.0386	0.039	1.07%	yes	0.0423	9.52%	yes	0.0424	9.8%	yes
$T_{Analysis}$	57	7.92	-86.1%	yes	0.749	-98.7%	yes	2.93	-94.9%	yes

Throughputs: ScenarioA 90 req/s ScenarioB 5 req/s										
	SimuCom	**SimQPN**			**LQNS**			**LQSIM**		
Metric	$\bar{x}$	$\bar{x}$	relDiff	statSig	$\bar{x}$	relDiff	statSig	$\bar{x}$	relDiff	statSig
U_{AS_CPU}	0.315	0.315	-	no	0.315	-	no	0.315	-	no
U_{AS_CPU}	0.204	0.204	-0.2%	yes	0.204	-	no	0.205	-	no
$R_{ScenarioA}$	0.00411	0.004	-2.6%	yes	0.00539	31.3%	yes	0.00549	33.7%	yes
$R_{ScenarioB}$	0.0389	0.039	0.151%	yes	0.0439	12.7%	yes	0.0442	13.4%	yes
$T_{Analysis}$	81.2	8.74	-89.2%	yes	0.85	-99%	yes	3.83	-95.3%	yes

Throughputs: ScenarioA 120 req/s ScenarioB 5 req/s										
	SimuCom	**SimQPN**			**LQNS**			**LQSIM**		
Metric	$\bar{x}$	$\bar{x}$	relDiff	statSig	$\bar{x}$	relDiff	statSig	$\bar{x}$	relDiff	statSig
U_{AS_CPU}	0.411	0.411	-	no	0.411	-	no	0.411	-	no
U_{AS_CPU}	0.222	0.222	-	no	0.222	-	no	0.222	-	no
$R_{ScenarioA}$	0.00409	0.004	-2.09%	yes	0.00614	50.2%	yes	0.0063	54.3%	yes
$R_{ScenarioB}$	0.0395	0.0392	-0.723%	yes	0.0457	15.8%	yes	0.0456	15.7%	yes
$T_{Analysis}$	106	9.48	-91%	yes	0.902	-99.1%	yes	4.71	-95.5%	yes

Throughputs: ScenarioA 150 req/s ScenarioB 5 req/s										
	SimuCom	**SimQPN**			**LQNS**			**LQSIM**		
Metric	$\bar{x}$	$\bar{x}$	relDiff	statSig	$\bar{x}$	relDiff	statSig	$\bar{x}$	relDiff	statSig
U_{AS_CPU}	0.507	0.507	-	no	0.507	-	no	0.507	-	no
U_{AS_CPU}	0.24	0.24	-	no	0.24	-	no	0.240	-	no
$R_{ScenarioA}$	0.00418	0.004	-4.31%	yes	0.00715	71.1%	yes	0.00745	78.2%	yes
$R_{ScenarioB}$	0.0405	0.0407	0.465%	yes	0.0479	18.3%	yes	0.0476	17.4%	yes
$T_{Analysis}$	132	10.3	-92.2%	yes	0.993	-99.2%	yes	5.83	-95.6%	yes

Table 8.9: ABB Demonstrator ScenarioA Variation Evaluation Results for 3000 sec

Throughputs: ScenarioA 30 req/s ScenarioB 5 req/s										
	SimuCom	**SimQPN**			**LQNS**			**LQSIM**		
Metric	$\bar{x}$	$\bar{x}$	relDiff	statSig	$\bar{x}$	relDiff	statSig	$\bar{x}$	relDiff	statSig
U_{AS_CPU}	0.123	0.123	0.224%	yes	0.123	-	no	0.123	-	no
U_{AS_CPU}	0.168	0.168	-	no	0.168	-	no	0.168	-	no
$R_{ScenarioA}$	0.00452	0.00493	9.16%	yes	0.00431	-4.62%	yes	0.0044	-2.71%	yes
$R_{ScenarioB}$	0.0383	0.038	-0.871%	yes	0.0409	6.65%	yes	0.041	7.03%	yes
$T_{Analysis}$	62.5	7.85	-87.4%	yes	0.905	-98.6%	yes	2.92	-95.3%	yes

Throughputs: ScenarioA 60 req/s ScenarioB 5 req/s										
	SimuCom	**SimQPN**			**LQNS**			**LQSIM**		
Metric	$\bar{x}$	$\bar{x}$	relDiff	statSig	$\bar{x}$	relDiff	statSig	$\bar{x}$	relDiff	statSig
U_{AS_CPU}	0.219	0.219	0.178%	yes	0.219	-	no	0.219	-	no
U_{AS_CPU}	0.186	0.186	-	no	0.186	-	no	0.186	-	no
$R_{ScenarioA}$	0.00419	0.004	-4.62%	yes	0.00477	13.8%	yes	0.00487	16.1%	yes
$R_{ScenarioB}$	0.0386	0.039	1.06%	yes	0.0423	9.52%	yes	0.0423	9.74%	yes
$T_{Analysis}$	111	9.29	-91.6%	yes	0.749	-99.3%	yes	4.85	-95.6%	yes

Throughputs: ScenarioA 90 req/s ScenarioB 5 req/s										
	SimuCom	**SimQPN**			**LQNS**			**LQSIM**		
Metric	$\bar{x}$	$\bar{x}$	relDiff	statSig	$\bar{x}$	relDiff	statSig	$\bar{x}$	relDiff	statSig
U_{AS_CPU}	0.315	0.315	0.0801%	yes	0.315	-	no	0.315	-	no
U_{AS_CPU}	0.204	0.204	-	no	0.204	-	no	0.204	-	no
$R_{ScenarioA}$	0.00411	0.004	-2.56%	yes	0.00539	31.4%	yes	0.0055	33.9%	yes
$R_{ScenarioB}$	0.0389	0.039	0.137%	yes	0.0439	12.6%	yes	0.0439	12.8%	yes
$T_{Analysis}$	161	11.2	-93%	yes	0.85	-99.5%	yes	6.45	-96%	yes

Throughputs: ScenarioA 120 req/s ScenarioB 5 req/s										
	SimuCom	**SimQPN**			**LQNS**			**LQSIM**		
Metric	$\bar{x}$	$\bar{x}$	relDiff	statSig	$\bar{x}$	relDiff	statSig	$\bar{x}$	relDiff	statSig
U_{AS_CPU}	0.411	0.411	0.072%	yes	0.411	-	no	0.411	-	no
U_{AS_CPU}	0.222	0.222	-	no	0.222	-	no	0.221	-	no
$R_{ScenarioA}$	0.00409	0.004	-2.14%	yes	0.00614	50.2%	yes	0.00631	54.4%	yes
$R_{ScenarioB}$	0.0395	0.0391	-0.986%	yes	0.0457	15.8%	yes	0.0454	15.2%	yes
$T_{Analysis}$	211	12.4	-94.1%	yes	0.902	-99.6%	yes	8.23	-96%	yes

Throughputs: ScenarioA 150 req/s ScenarioB 5 req/s										
	SimuCom	**SimQPN**			**LQNS**			**LQSIM**		
Metric	$\bar{x}$	$\bar{x}$	relDiff	statSig	$\bar{x}$	relDiff	statSig	$\bar{x}$	relDiff	statSig
U_{AS_CPU}	0.507	0.507	0.0412%	yes	0.507	-	no	0.507	-	no
U_{AS_CPU}	0.24	0.24	-	no	0.24	-	no	0.24	-	no
$R_{ScenarioA}$	0.00418	0.004	-4.29%	yes	0.00715	71.2%	yes	0.00743	77.8%	yes
$R_{ScenarioB}$	0.0405	0.0407	0.542%	yes	0.0479	18.3%	yes	0.0479	18.3%	yes
$T_{Analysis}$	262	13.9	-94.7%	yes	0.993	-99.6%	yes	10.0	-96.2%	yes

Table 8.10: ABB Demonstrator ScenarioA Variation Evaluation Results for 6000 sec

Throughputs: ScenarioA 30 req/s ScenarioB 3 req/s									
	SimuCom	**SimQPN**			**LQNS**			**LQSIM**		
Metric	$\bar{x}$	$\bar{x}$	relDiff	statSig	$\bar{x}$	relDiff	statSig	$\bar{x}$	relDiff	statSig
U_{AS_CPU}	0.114	0.114	0.0699%	yes	0.114	-	no	0.114	-	no
U_{AS_CPU}	0.108	0.108	-	no	0.108	-	no	0.108	-	no
$R_{ScenarioA}$	0.00428	0.004	-6.53%	yes	0.00423	-1.19%	yes	0.00431	0.788%	yes
$R_{ScenarioB}$	0.039	0.039	0.0966%	yes	0.0385	-1.31%	yes	0.0386	-	no
$T_{Analysis}$	16.6	6.61	-60.2%	yes	0.688	-95.9%	yes	1.41	-91.5%	yes

Throughputs: ScenarioA 30 req/s ScenarioB 4 req/s										
	SimuCom	**SimQPN**			**LQNS**			**LQSIM**		
Metric	$\bar{x}$	$\bar{x}$	relDiff	statSig	$\bar{x}$	relDiff	statSig	$\bar{x}$	relDiff	statSig
U_{AS_CPU}	0.118	0.118	-0.147%	yes	0.118	-	no	0.118	-	no
U_{AS_CPU}	0.138	0.138	-	no	0.138	-	no	0.138	-	no
$R_{ScenarioA}$	0.00417	0.004	-4.15%	yes	0.00427	2.29%	yes	0.00437	4.6%	yes
$R_{ScenarioB}$	0.0372	0.037	-0.524%	yes	0.0396	6.54%	yes	0.0395	6.28%	yes
$T_{Analysis}$	16.9	6.65	-60.7%	yes	0.688	-96%	yes	1.42	-91.6%	yes

Throughputs: ScenarioA 30 req/s ScenarioB 5 req/s										
	SimuCom	**SimQPN**			**LQNS**			**LQSIM**		
Metric	$\bar{x}$	$\bar{x}$	relDiff	statSig	$\bar{x}$	relDiff	statSig	$\bar{x}$	relDiff	statSig
U_{AS_CPU}	0.123	0.123	0.196%	yes	0.123	-	no	0.122	-	no
U_{AS_CPU}	0.168	0.168	-	no	0.168	-	no	0.167	-	no
$R_{ScenarioA}$	0.00452	0.00477	5.42%	yes	0.00431	-4.67%	yes	0.00437	-3.37%	yes
$R_{ScenarioB}$	0.0383	0.038	-0.87%	yes	0.0409	6.65%	yes	0.0408	6.56%	yes
$T_{Analysis}$	18	6.83	-62%	yes	0.905	-95%	yes	1.48	-91.8%	yes

Throughputs: ScenarioA 30 req/s ScenarioB 10 req/s										
	SimuCom	**SimQPN**			**LQNS**			**LQSIM**		
Metric	$\bar{x}$	$\bar{x}$	relDiff	statSig	$\bar{x}$	relDiff	statSig	$\bar{x}$	relDiff	statSig
U_{AS_CPU}	0.144	0.144	-0.117%	yes	0.144	-	no	0.145	-	no
U_{AS_CPU}	0.318	0.318	-	no	0.318	-	no	0.320	-	no
$R_{ScenarioA}$	0.00513	0.005	-2.50%	yes	0.00455	-11.2%	yes	0.00466	-9.2%	yes
$R_{ScenarioB}$	0.0379	0.038	0.362%	yes	0.0487	28.7%	yes	0.0494	30.6%	yes
$T_{Analysis}$	19.4	6.8	-64.9%	yes	0.81	-95.8%	yes	1.52	-92.2%	yes

Table 8.11: ABB Demonstrator ScenarioB Variation Evaluation Results for 1500 sec

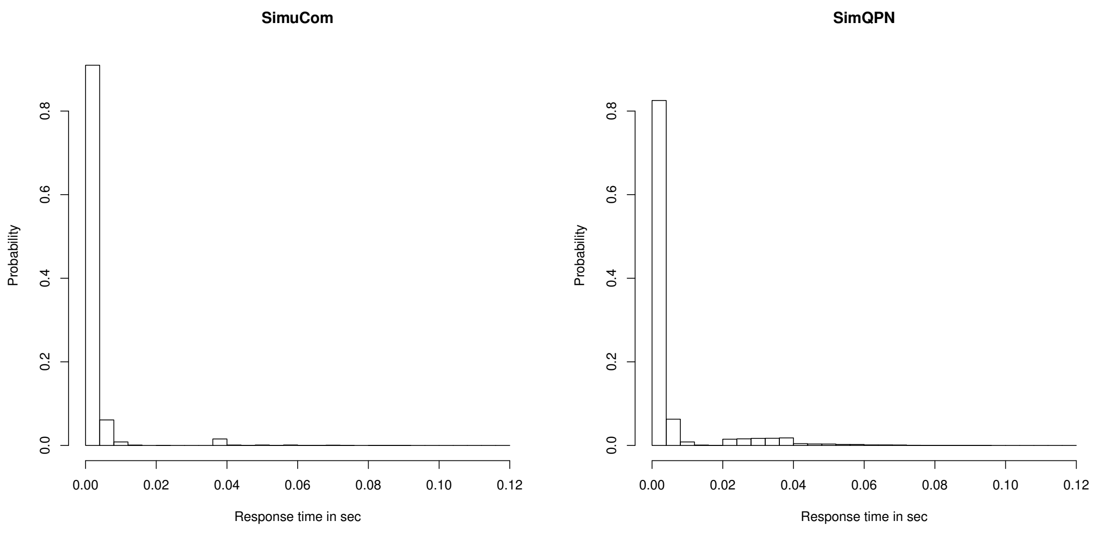


Figure 8.5: ABB Demonstrator SimuCom vs SimQPN Histograms
ScenarioA 120/5 req/s 6000sec

Throughputs: ScenarioA 30 req/s ScenarioB 3 req/s										
	SimuCom	**SimQPN**			**LQNS**			**LQSIM**		
Metric	$\bar{x}$	$\bar{x}$	relDiff	statSig	$\bar{x}$	relDiff	statSig	$\bar{x}$	relDiff	statSig
U_{AS_CPU}	0.114	0.114	0.0433%	yes	0.114	-	no	0.114	-	no
U_{AS_CPU}	0.108	0.108	-	no	0.108	-	no	0.108	-	no
$R_{ScenarioA}$	0.00428	0.004	-6.55%	yes	0.00423	-1.21%	yes	0.00431	0.586%	yes
$R_{ScenarioB}$	0.039	0.039	0.0985%	yes	0.0385	-1.31%	yes	0.0384	-1.34%	yes
$T_{Analysis}$	30.5	6.8	-77.7%	yes	0.688	-97.7%	yes	1.83	-94%	yes

Throughputs: ScenarioA 30 req/s ScenarioB 4 req/s										
	SimuCom	**SimQPN**			**LQNS**			**LQSIM**		
Metric	$\bar{x}$	$\bar{x}$	relDiff	statSig	$\bar{x}$	relDiff	statSig	$\bar{x}$	relDiff	statSig
U_{AS_CPU}	0.118	0.118	-0.213%	yes	0.118	-	no	0.118	-	no
U_{AS_CPU}	0.138	0.138	-	no	0.138	-	no	0.138	-	no
$R_{ScenarioA}$	0.00418	0.004	-4.2%	yes	0.00427	2.23%	yes	0.00434	3.98%	yes
$R_{ScenarioB}$	0.0372	0.037	-0.534%	yes	0.0396	6.53%	yes	0.0397	6.68%	yes
$T_{Analysis}$	31.5	7.1	-77.4%	yes	0.688	-97.8%	yes	1.85	-94.1%	yes

Throughputs: ScenarioA 30 req/s ScenarioB 5 req/s										
	SimuCom	**SimQPN**			**LQNS**			**LQSIM**		
Metric	$\bar{x}$	$\bar{x}$	relDiff	statSig	$\bar{x}$	relDiff	statSig	$\bar{x}$	relDiff	statSig
U_{AS_CPU}	0.123	0.123	0.241%	yes	0.123	-	no	0.122	-	no
U_{AS_CPU}	0.168	0.168	-	no	0.168	-	no	0.168	-	no
$R_{ScenarioA}$	0.00452	0.0049	8.32%	yes	0.00431	-4.71%	yes	0.00437	-3.34%	yes
$R_{ScenarioB}$	0.0383	0.038	-0.872%	yes	0.0409	6.65%	yes	0.0409	6.69%	yes
$T_{Analysis}$	32.4	7.15	-78%	yes	0.905	-97.2%	yes	1.99	-93.9%	yes

Throughputs: ScenarioA 30 req/s ScenarioB 10 req/s										
	SimuCom	**SimQPN**			**LQNS**			**LQSIM**		
Metric	$\bar{x}$	$\bar{x}$	relDiff	statSig	$\bar{x}$	relDiff	statSig	$\bar{x}$	relDiff	statSig
U_{AS_CPU}	0.144	0.144	-0.117%	yes	0.144	-	no	0.145	0.373%	yes
U_{AS_CPU}	0.318	0.318	-	no	0.318	-	no	0.319	-	no
$R_{ScenarioA}$	0.00513	0.005	-2.5%	yes	0.00455	-11.2%	yes	0.00467	-9.02%	yes
$R_{ScenarioB}$	0.0379	0.038	0.364%	yes	0.0487	28.7%	yes	0.0491	29.7%	yes
$T_{Analysis}$	37.3	7.46	-80%	yes	0.81	-97.8%	yes	2.13	-94.3%	yes

Table 8.12: ABB Demonstrator ScenarioB Variation Evaluation Results for 3000 sec

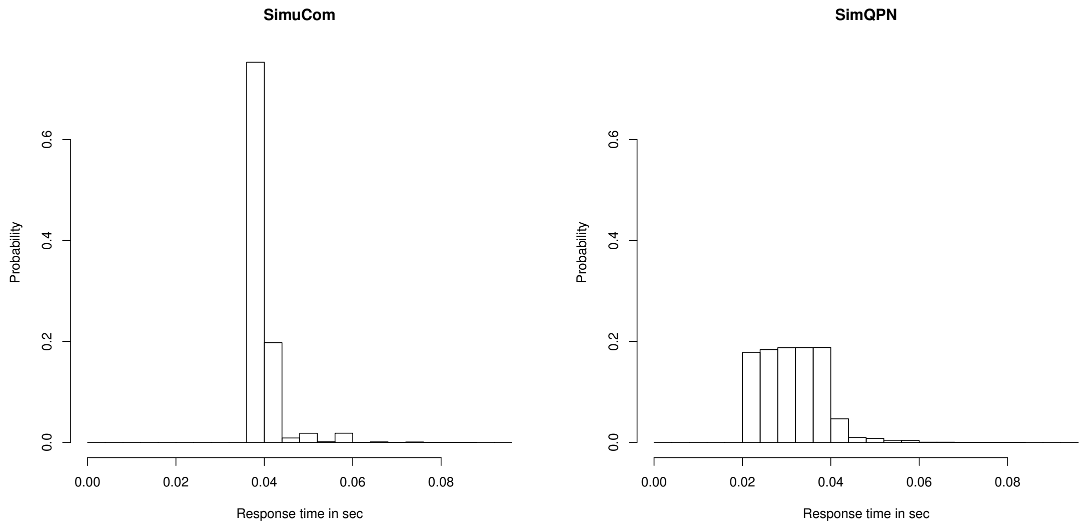


Figure 8.6: ABB Demonstrator SimuCom vs SimQPN Histograms
ScenarioB 120/5 req/s 6000sec

Throughputs: ScenarioA 30 req/s ScenarioB 3 req/s									
	SimuCom	**SimQPN**			**LQNS**			**LQSIM**	
Metric	$\bar{x}$	$\bar{x}$	relDiff	statSig	$\bar{x}$	relDiff	statSig	$\bar{x}$	relDiff	statSig
U_{AS_CPU}	0.114	0.114	0.0417%	yes	0.114	-0.0223%	yes	0.114	-	no
U_{AS_CPU}	0.108	0.108	-	no	0.108	-	no	0.109	-	no
$R_{ScenarioA}$	0.00428	0.004	-6.57%	yes	0.00423	-1.23%	yes	0.00431	0.67%	yes
$R_{ScenarioB}$	0.039	0.039	0.0963%	yes	0.0385	-1.31%	yes	0.0386	-1.03%	yes
$T_{Analysis}$	59.9	7.74	-87%	yes	0.688	-98.9%	yes	2.83	-95.3%	yes

Throughputs: ScenarioA 30 req/s ScenarioB 4 req/s										
	SimuCom	**SimQPN**			**LQNS**			**LQSIM**		
Metric	$\bar{x}$	$\bar{x}$	relDiff	statSig	$\bar{x}$	relDiff	statSig	$\bar{x}$	relDiff	statSig
U_{AS_CPU}	0.118	0.118	-0.220%	yes	0.118	-	no	0.118	-	no
U_{AS_CPU}	0.138	0.138	-0.113%	yes	0.138	-	no	0.138	-	no
$R_{ScenarioA}$	0.00417	0.004	-4.18%	yes	0.00427	2.26%	yes	0.00434	3.92%	yes
$R_{ScenarioB}$	0.0372	0.037	-0.536%	yes	0.0396	6.53%	yes	0.0398	6.87%	yes
$T_{Analysis}$	61.5	7.76	-87.4%	yes	0.688	-98.9%	yes	2.88	-95.3%	yes

Throughputs: ScenarioA 30 req/s ScenarioB 5 req/s										
	SimuCom	**SimQPN**			**LQNS**			**LQSIM**		
Metric	$\bar{x}$	$\bar{x}$	relDiff	statSig	$\bar{x}$	relDiff	statSig	$\bar{x}$	relDiff	statSig
U_{AS_CPU}	0.123	0.123	0.224%	yes	0.123	-	no	0.123	-	no
U_{AS_CPU}	0.168	0.168	-	no	0.168	-	no	0.168	-	no
$R_{ScenarioA}$	0.00452	0.00493	9.16%	yes	0.00431	-4.62%	yes	0.0044	-2.71%	yes
$R_{ScenarioB}$	0.0383	0.038	-0.871%	yes	0.0409	6.65%	yes	0.041	7.03%	yes
$T_{Analysis}$	62.5	7.85	-87.4%	yes	0.905	-98.6%	yes	2.92	-95.3%	yes

Throughputs: ScenarioA 30 req/s ScenarioB 10 req/s										
	SimuCom	**SimQPN**			**LQNS**			**LQSIM**		
Metric	$\bar{x}$	$\bar{x}$	relDiff	statSig	$\bar{x}$	relDiff	statSig	$\bar{x}$	relDiff	statSig
U_{AS_CPU}	0.144	0.144	-0.117%	yes	0.144	-	no	0.144	-	no
U_{AS_CPU}	0.318	0.318	-	no	0.318	-	no	0.317	-	no
$R_{ScenarioA}$	0.00513	0.005	-2.49%	yes	0.00455	-11.2%	yes	0.00465	-9.28%	yes
$R_{ScenarioB}$	0.0379	0.038	0.362%	yes	0.0487	28.7%	yes	0.0489	29.3%	yes
$T_{Analysis}$	71.2	8.05	-88.7%	yes	0.81	-98.9%	yes	3.19	-95.5%	yes

Table 8.13: ABB Demonstrator ScenarioB Variation Evaluation Results for 6000 sec

The distributions look almost alike and fit the relatively low errors of the mean response times.

The distribution for scenario B with the same workload and logical runtime (Figure 8.6) is less accurate. SimuCom predicts, that there are almost no requests faster than 0.03 seconds. SimQPN predicts quite a big number of requests as fast as 0.02 seconds. 0.04 seconds and upwards the distributions look similar again. The most likely reason for this is that the ABB model contains branches with very low branch probabilities and very low resource demands. With response time differences between the diagrams below 20ms, even small rounding errors or slight differences in how the random number generators work could cause significant differences.

A simulation run with 18000 logical seconds produced the exact same distributions, the error is not due to a lack of data.

The SimQPN simulation run for 6000 seconds with probes enabled took about 1.3 times as long as without probes. In this case, the SimQPN run was about 15 times faster than the SimuCom run.

8.5 Case Study 3: MediaStore

8.5.1 Overview

The MediaStore is an example model that does not truly reach the complexity of an industrial system. It is still included as it is still much larger than the simple models used to evaluate individual features. The PCM instance used for this thesis is based on the MediaStore scenario used in [Koz08, pp. 237]. It was ported to the current PCM version. As the results presented in [Koz08] could not be reproduced, we either deal with a slightly different version or the tools behave differently in the current version. As the focus of this evaluation is to compare to the SimuCom reference values, rather than to interpret the results at a domain level, this mismatch is of no concern for this discussion.

For this evaluation, only a single scenario with a closed workload is analyzed. In the following, it is referred to as scenario A.

A number of test runs using SimuCom and the relative stopping criterion for the scenario mean response time finished after close to 34000 logical seconds, which is taken as the reference time T.

8.5.2 Complexity

The MediaStore PCM instance is very simple. In the usage model it has one *UsageScenario* with a single *EntryLevelSystemCall*, no loops and no branches. the *UsageScenario* has a closed workload with one user and 0 think time.

The system part references 5 *AssemblyContexts* in 5 *BasicComponents*. 4 *ExternalCallActions* are referenced. No branches are used. Two loops are present (one *LoopAction* and one *CollectionIteratorAction*). No *CompositeComponents*, *SubSystems* and no *PassiveResources* are referenced.

For the stochastic expressions the commonly used IntPMF and DoublePDF constructs are used.

1 user, 0 think time											
	SimuCom	**SimQPN**			**LQNS**			**LQSIM**			
Metric	$\bar{x}$	$\bar{x}$	relDiff	statSig	$\bar{x}$	relDiff	statSig	$\bar{x}$	relDiff	statSig	
U_{AS_CPU}	0.359	0.362	1.02%	yes	0.363	1.20%	yes	-	-	-	
U_{DB_CPU}	0.641	0.638	-0.573%	yes	0.638	-0.525%	yes	-	-	-	
$R_{Scenario}$	8.37	8.28	-1.05%	yes	7.51	-10.2%	yes	-	-	-	
$X_{Scenario}$	0.120	0.121	1.1%	yes	0.129	7.92%	yes	-	-	-	
$T_{Analysis}$	4.2	1.96	-53.4%	yes	0.361	-91.4%	yes	-	-	-	

Table 8.14: MediaStore ScenarioA Evaluation Results for 17000 sec

1 user, 0 think time											
	SimuCom	**SimQPN**			**LQNS**			**LQSIM**			
Metric	$\bar{x}$	$\bar{x}$	relDiff	statSig	$\bar{x}$	relDiff	statSig	$\bar{x}$	relDiff	statSig	
U_{AS_CPU}	0.358	0.362	1.36%	yes	0.363	1.48%	yes	-	-	-	
U_{DB_CPU}	0.642	0.638	-0.76%	yes	0.638	-0.68%	yes	-	-	-	
$R_{Scenario}$	8.39	8.28	-1.28%	yes	7.51	-10.5%	yes	-	-	-	
$X_{Scenario}$	0.119	0.121	1.27%	yes	0.129	8.19%	yes	-	-	-	
$T_{Analysis}$	5.68	1.89	-66.7%	yes	0.361	-93.6%	yes	-	-	-	

Table 8.15: MediaStore ScenarioA Evaluation Results for 34000 sec

1 user, 0 think time											
	SimuCom	**SimQPN**			**LQNS**			**LQSIM**			
Metric	$\bar{x}$	$\bar{x}$	relDiff	statSig	$\bar{x}$	relDiff	statSig	$\bar{x}$	relDiff	statSig	
U_{AS_CPU}	0.358	0.363	1.31%	yes	0.363	1.34%	yes	-	-	-	
U_{DB_CPU}	0.642	0.637	-0.73%	yes	0.638	-0.599%	yes	-	-	-	
$R_{Scenario}$	8.38	8.28	-1.12%	yes	7.51	-10.3%	yes	-	-	-	
$X_{Scenario}$	0.119	0.121	1.14%	yes	0.129	8.05%	yes	-	-	-	
$T_{Analysis}$	8.88	2.02	-77.3%	yes	0.361	-96%	yes	-	-	-	

Table 8.16: MediaStore ScenarioA Evaluation Results for 68000 sec

8.5.3 Results

Table 8.14, Table 8.15 and Table 8.16 show the results for scenario A for each of the three logical simulation times (17000, 34000 and 68000 seconds). It is unclear why the LQSim solver crashed during the execution. It is possible that there are issues with the newer version which is used compared to the version in [Koz08]. In practice, the LQNS results are more important because of the very low execution time.

The results between SimuCom, SimQPN and LQNS do not differ by much. It is interesting that the processing resource utilizations in this case are predicted almost identically by SimQPN and LQNS. The error stays under 1.5% but is statistically significant. It looks like both approaches employ a similar set of abstractions. One possibility would be that this error is introduced by the DependencySolver pre-processing, which both approaches employ.

Regarding the scenario mean response time and throughput predictions the SimQPN and LQNS solvers differ. SimQPN reaches a much smaller error which stays below 1.3% for both metrics. LQNS reaches an acceptable error for the mean response time of just over 10% and a less acceptable error for the throughput of about 8%.

There are no notable differences in the results between the different logical runtimes.

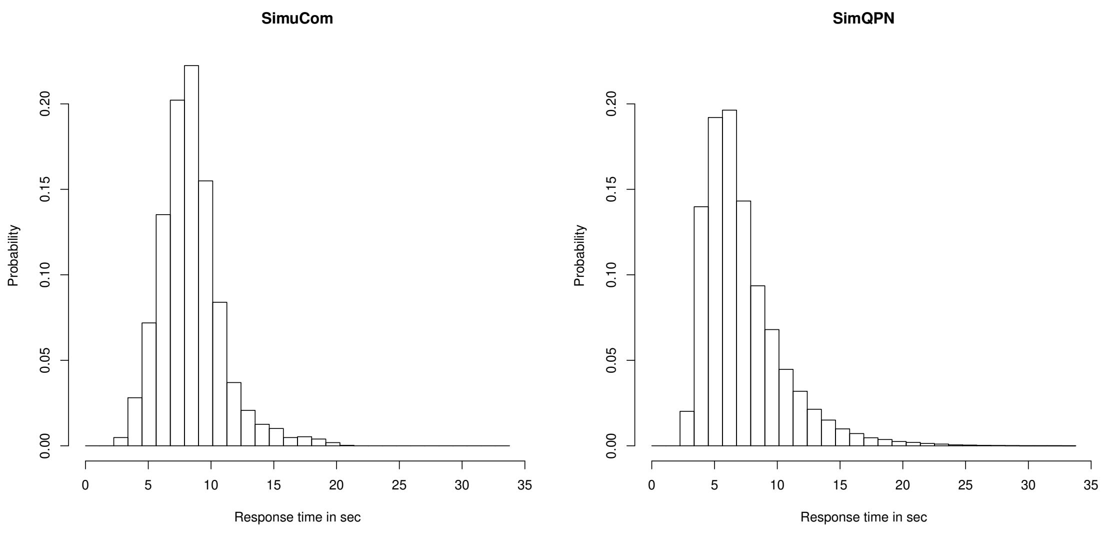


Figure 8.7: MediaStore SimuCom vs SimQPN Histograms
ScenarioA 68000sec

Figure 8.7 shows the response time distributions for scenario A and a logical simulation time of 68000 seconds. The distributions show a good match. The SimQPN distribution is slightly more steep on the left end and reaches out a little further to the right. It appears that the error introduced by the loop mapping (see Section 5.2.5) is offset by the fact that the DoublePDFs used inside the internal action resource demand specifications are reduced to mean values, reducing the spread.

The simulation run times for this simple scenario are below 10 seconds for both SimuCom and SimQPN, which is too low for a good comparison as the initialization time makes up most of the execution time.

8.6 Case Study 4: CoCoME

8.6.1 Overview

CoCoME stands for Common Component Modeling Example and describes a trading system as it can be observed in a supermarket handling sales [HKW+08]. For this evaluation a model representing the CoCoME example was available. We assume that it meets our requirements of size and complexity.

The model was migrated and therefore might not match exactly the description in [HKW+08]. The workload has also been adapted so that the system is not overloaded and reaches a steady state. As we focus on comparing the SimuCom results to the other solvers this is of no concern for this evaluation.

A single scenario with a closed workload is analyzed. In the following, it will be referred to as scenario A.

A test run using SimuCom and a relative stopping criterion for the mean response time showed that this scenario requires a very large logical simulation time. Only after 600000 simulated seconds did the run finish. This time was taken as the base time T.

8.6.2 Complexity

The CoCoME PCM instance has a very simple usage model. It has one *UsageScenario* with a single *EntryLevelSystemCall*, no loops and no branches. The closed workload in this case has 20 users and a think time of 30 seconds.

The system part of the instance is of larger than average size and is dominated by loops. The high number of loops cause the high logical simulation time as each requests spends considerable amount of time in the system, keeping the simulation engine busy through many loop events. 13 *AssemblyContext*s are referenced in 10 *BasicComponent*s and 3 *CompositeComponent*s. 14 *ExternalCallAction*s, 3 branches and 6 loops (*LoopAction*s) are referenced. No *SubSystem*s and no *PassiveResource*s are referenced.

The stochastic expressions do not use more advanced constructs than the commonly used IntPMF and DoublePDF.

8.6.3 Results

20 users, 30 sec think time										
	SimuCom	**SimQPN**			**LQNS**			**LQSIM**		
Metric	$\bar{x}$	$\bar{x}$	relDiff	statSig	$\bar{x}$	relDiff	statSig	$\bar{x}$	relDiff	statSig
U_{AS_CPU}	0.642	0.643	0.193%	yes	0.646	0.555%	yes	-	-	-
$R_{Scenario}$	2.65	2.59	-2.27%	yes	2.47	-6.9%	yes	-	-	-
$X_{Scenario}$	0.613	0.614	0.176%	yes	0.616	0.553%	yes	-	-	-
$T_{Analysis}$	1250	594	-52.5%	yes	5.73	-99.5%	yes	-	-	-

Table 8.17: CoCoME ScenarioA Evaluation Results for 300000 sec

20 users, 30 sec think time										
	SimuCom	**SimQPN**			**LQNS**			**LQSIM**		
Metric	$\bar{x}$	$\bar{x}$	relDiff	statSig	$\bar{x}$	relDiff	statSig	$\bar{x}$	relDiff	statSig
U_{AS_CPU}	0.644	0.645	-	no	0.646	0.281%	yes	-	-	-
$R_{Scenario}$	2.57	2.54	-	no	2.47	-3.69%	yes	-	-	-
$X_{Scenario}$	0.614	0.615	-	no	0.616	0.284%	yes	-	-	-
$T_{Analysis}$	2488	1254	-49.6%	yes	5.73	-99.8%	yes	-	-	-

Table 8.18: CoCoME ScenarioA Evaluation Results for 600000 sec

20 users, 30 sec think time										
	SimuCom	**SimQPN**			**LQNS**			**LQSIM**		
Metric	$\bar{x}$	$\bar{x}$	relDiff	statSig	$\bar{x}$	relDiff	statSig	$\bar{x}$	relDiff	statSig
U_{AS_CPU}	0.645	0.645	-	no	0.646	0.176%	yes	-	-	-
$R_{Scenario}$	2.53	2.52	-	no	2.47	-2.41%	yes	-	-	-
$X_{Scenario}$	0.615	0.615	-	no	0.616	0.182%	yes	-	-	-
$T_{Analysis}$	4931	2387	-51.6%	yes	5.73	-99.9%	yes	-	-	-

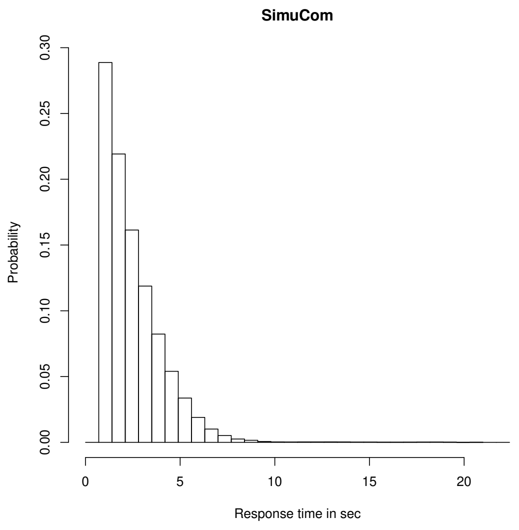
Table 8.19: CoCoME ScenarioA Evaluation Results for 1200000 sec

Table 8.17, Table 8.18 and Table 8.19 show the results for scenario A for each of the three logical simulation times (300000, 600000 and 1200000 seconds). No data is available for LQSim as the solver crashed for unknown reasons.

In this scenario we notice an improvement of the prediction results with an increased runtime. The system appears to reach its steady state quite late and with an increased runtime the impact of the error at the beginning of the simulation is reduced.

At 300000 simulated seconds, SimQPN shows a small significant error for the processing resource utilization, scenario mean response time and throughput. The error for the mean response time is below 2.5%, the other errors are below 0.5%. From 600000 seconds on the SimQPN and SimuCom solvers show no statistically significant differences.

The utilization and throughput are predicted almost equally well by LQNS. The error goes from just over 0.5% at 300000 seconds to less then 0.2% at 1200000 seconds. The mean response time shows a slighly higher error, going from almost 7% at 300000 seconds to below 2.5% at 1200000 seconds. In practice, this is a very good result as well, especially for the reduction in execution time of almost two orders of magnitude.



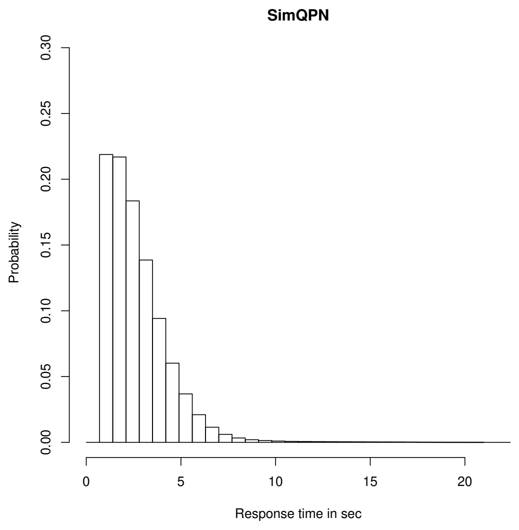


Figure 8.8: CoCoME SimuCom vs SimQPN Histograms
ScenarioA 1200000sec

Figure 8.8 shows the response time distributions for scenario A and a logical simulation time of 1200000 seconds. The distributions show a very good match. The SimQPN distribution has a slightly lower peak and more values to the right of the peak. It appears that the error introduced by the loop mapping (see Section 5.2.5) is offset by the fact that the DoublePDFs and IntPMFs used inside the internal action resource demand specifications are reduced to mean values, reducing the spread.

As this is the scenario with the highest simulation execution times, a detailed examination is useful. The SimQPN simulation run with probes enabled took 3572 seconds, which is 1.5 times as long as the mean execution time of a run without probes (2382 seconds). This time, the SimQPN run is only about 1.4 times faster than the SimuCom run. The reason might be that this scenario is dominated by loops, which require a relatively high number of places, transitions and colors to map (see Section 5.2.5).

8.7 Case Study 5: Business Reporting System (BRS)

8.7.1 Overview

The Business Reporting System (BRS) model is taken from [MKBR10]. It was created to evaluate a method of automatic software architecture optimization. Apart from migrating the model to the current PCM version, the workload and processing resource processing rates were adapted until the system reached a steady state during simulation.

A single usage scenario with an open workload and a throughput of 20 requests/s was analyzed for this case study. It is denoted as scenario A in the following.

A number of test runs using SimuCom and the relative stopping criterion for the mean response time finished after only 200 simulated seconds. That time was taken as the base time T.

8.7.2 Complexity

The BRS scenario PCM instance is of a high complexity compared to the other case study instances. The usage model has one *UsageScenario* with 9 *EntryLevelSystemCall*s and 2 loops. It contains no branches. It has an open workload with an arrival rate of 20 requests/s.

The system part of the instance is above average size as well. 13 *AssemblyContext*s are referenced in 9 *BasicComponent*s and 2 *CompositeComponent*s. 28 *ExternalCallAction*s, 5 loops (*LoopAction*s) and 5 branches are referenced. No *SubSystem*s and no *PassiveResource*s are referenced.

The most advanced constructs used in the stochastic expressions are EnumPDF in the usage model and DoublePDF in the SEFFs.

8.7.3 Results

Throughput: 20 req/s										
	SimuCom	**SimQPN**			**LQNS**			**LQSIM**		
Metric	$\bar{x}$	$\bar{x}$	relDiff	statSig	$\bar{x}$	relDiff	statSig	$\bar{x}$	relDiff	statSig
$U_{Server1\_CPU}$	0.09	0.0906	0.607%	yes	0.09	-0.0243%	yes	-	-	-
$U_{Server2\_CPU}$	0.413	0.415	0.524%	yes	0.413	-	no	-	-	-
$U_{Server3\_CPU}$	0.149	0.149	-	no	0.149	-	no	-	-	-
$U_{Server4\_CPU}$	0.0471	0.0471	-	no	0.0471	-	no	-	-	-
$R_{Scenario}$	0.0350	0.0393	12.3%	yes	0.0509	45.7%	yes	-	-	-
$T_{Analysis}$	49.8	8.1	-83.8%	yes	9.32	-81.3%	yes	-	-	-

Table 8.20: BRS ScenarioA Evaluation Results for 100 sec

Table 8.20, Table 8.21 and Table 8.22 show the results for scenario A for each of the three logical simulation times (100, 200 and 400 seconds). For unknown reasons, LQSim crashed and no data is available. LQNS showed no problems in execution.

The processing resource utilizations are predicted almost perfectly by SimQPN and LQNS. There is either no statistically significant difference, or the error is under 1%. The mean

Throughput: 20 req/s										
	SimuCom	**SimQPN**			**LQNS**			**LQSIM**		
Metric	$\bar{x}$	$\bar{x}$	relDiff	statSig	$\bar{x}$	relDiff	statSig	$\bar{x}$	relDiff	statSig
$U_{Server1\_CPU}$	0.09	0.09	-	no	0.09	-0.0134%	yes	-	-	-
$U_{Server2\_CPU}$	0.413	0.413	-	no	0.413	-0.0106%	yes	-	-	-
$U_{Server3\_CPU}$	0.148	0.149	-	no	0.149	-	no	-	-	-
$U_{Server4\_CPU}$	0.0471	0.047	-	no	0.0471	-	no	-	-	-
$R_{Scenario}$	0.0349	0.039	11.7%	yes	0.0509	45.7%	yes	-	-	-
$T_{Analysis}$	95	10.7	-88.7%	yes	9.32	-90.2%	yes	-	-	-

Table 8.21: BRS ScenarioA Evaluation Results for 200 sec

Throughput: 20 req/s										
	SimuCom	**SimQPN**			**LQNS**			**LQSIM**		
Metric	$\bar{x}$	$\bar{x}$	relDiff	statSig	$\bar{x}$	relDiff	statSig	$\bar{x}$	relDiff	statSig
$U_{Server1\_CPU}$	0.09	0.0899	-	no	0.09	-0.00759%	yes	-	-	-
$U_{Server2\_CPU}$	0.413	0.413	-	no	0.413	-0.00634%	yes	-	-	-
$U_{Server3\_CPU}$	0.149	0.148	-	no	0.149	-	no	-	-	-
$U_{Server4\_CPU}$	0.0471	0.047	-	no	0.0471	-	no	-	-	-
$R_{Scenario}$	0.0350	0.0389	11.2%	yes	0.0509	45.7%	yes	-	-	-
$T_{Analysis}$	183	16.3	-91.1%	yes	9.32	-95%	yes	-	-	-

Table 8.22: BRS ScenarioA Evaluation Results for 400 sec

response time of the scenario is predicted to a satisfying degree by SimQPN. Slightly increasing in accuracy with an increased simulation time, it approaches an error of 11%. LQNS, on the other hand, has an error of over 45%. It is unclear by how much this value could have been improved by tuning the LQNS configuration.

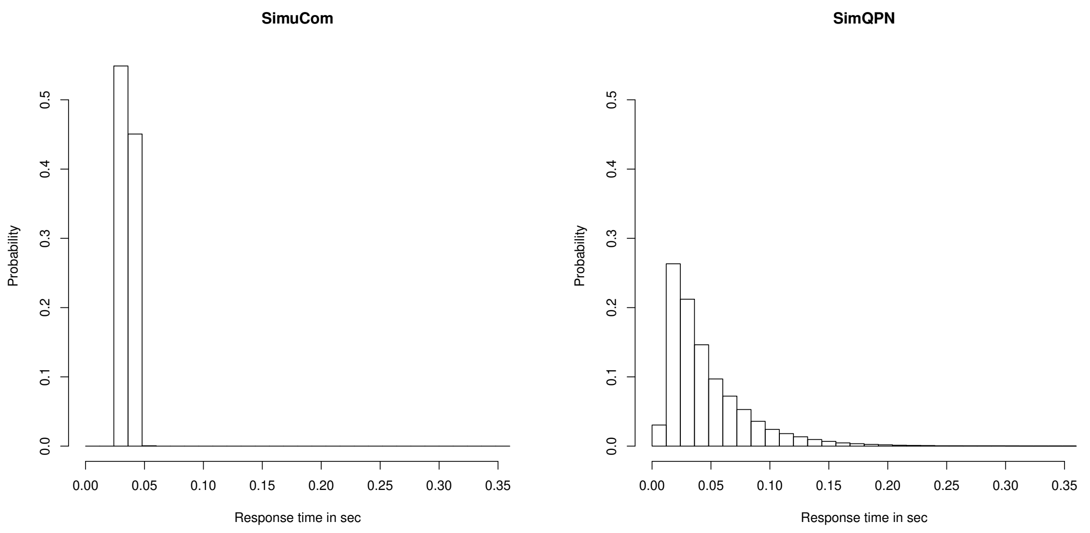


Figure 8.9: BRS SimuCom vs SimQPN Histograms
ScenarioA 400sec

Figure 8.9 shows the response time distributions for scenario A and a logical simulation time of 400 seconds. We can clearly see the spread introduced by the approximations in the loop mapping (see Section 5.2.5). Even though DoublePDFs are used for internal action resource demand specifications, the actual values show very little variation (e.g., 'DoublePDF[(0.2;0.1)(0.3;0.6)(0.4;0.3)]'). Therefore the SimuCom histogram shows very little spread and there is little variance in the resource demands to offset the spread introduced by the loops (as encountered in Section 8.5 and Section 8.6).

8.8 Summary

The case studies show very satisfying results. Even though a numerous range of scenarios with different complexities were evaluated, the SimQPN solver predicted all mean value metrics with high accuracy. At the same time, the analysis overhead compared to SimuCom could be significantly reduced, in many cases by an order of magnitude. Models with a high number of loops showed a higher analysis overhead, but remained below 60% of the SimuCom analysis time.

LQSim could only handle the ABB Demonstrator case study, where it showed equal results to LQNS for all metrics. As LQNS runs much faster than LQSim and could handle all but the SPECjAppServer2004_Next case study, LQNS can be recommended over LQSim.

Looking at the individual metrics, processing resource utilizations predictions showed very little differences between the solvers. For usage scenario throughputs, SimQPN showed predictions within 2% of the SimuCom results. LQNS showed an error of up to 10%. Mean response time predictions by SimQPN showed an error of up to 15% compared to SimuCom. The error was independent of the workload. This was not the case with LQNS. In the ABB Demonstrator case study, the error increased from about 10% to over 70% with an increasing workload.

LQNS, being an analytical method, generally runs about an order of magnitude faster than the SimQPN solver. It can therefore be recommended for utilization analysis. For throughput analysis it can be recommended if the margin for an acceptable error is larger than 10%. For mean response time analysis it can not be recommended. SimQPN can be recommended for all mean value metric analyses.

Response time distributions can normally not be computed with QPNs as the identity of individual tokens is not known. An experimental extension of SimQPN allows to track individual tokens to a certain degree. This makes a comparison with distributions provided by SimuCom possible. The abstractions made when tracking tokens cause distributions of varying accuracy. Loops caused the highest variation in response times. Overall the results were still satisfying. Even when the distributions show a higher spread a performance analyst can still draw information from the distribution like the number of peaks and the general shape of the distribution. When generating response time distributions, the runtime of the SimQPN solver was between 1.4 and 15 times faster than SimuCom.

The SPECjAppServer2004_Next case study helped to identify a minor shortcoming with the employed SimQPN version, as the support for empirical distributions in queueing places did not satisfy the requirements of the PCM-to-QPN mapping. Initially the use of the Exp function in the inter-arrival times of the workload specification caused a significant error with the mean response time predictions. A manual workaround resolved the prediction errors, showing that this was simply an implementation issue which will be resolved in future versions of the SimQPN solver. The SPECjAppServer2004_Next case study also showed that the implementation handling the folding of distribution functions (which is part of the PCM-Bench) needs to be improved. In this case it introduced an error of 5%.

9. Summary and Future Work

This chapter provides a summary of the contributions of the thesis and an overview of future work.

9.1 Contributions

In this thesis, a formal mapping of the Palladio Component Model (PCM) to Queueing Petri Nets (QPNs) was developed. A solver tool implementing the mapping transformation was developed and used to evaluate the mapping both feature by feature and for a number of case studies of realistic size and complexity. The solver tool was compared to the existing reference solver SimuCom, as well as to a set of solvers based on a transformation to Layered Queueing Networks (LQNs): LQNS and LQSim. The main evaluation criteria were results accuracy and analysis overhead.

The mapping builds on top of the DependencySolver, a module in PCM-Bench implementing a pre-processing step on the PCM instance, eliminating variables in stochastic expressions and providing context management functionality needed for model traversal. The number of combinations of PCM meta-model entities and stochastic expressions language entities can quickly explode. Therefore, the scope of the thesis was limited to the correct mapping of the meta-model entities, assuming the DependencySolver correctly handles the common stochastic expressions typically used in real-life models.

It was possible to map all PCM entities to QPN elements. Regarding mean value metric predictions, limitations exist only for less common entities not encountered in any of the case studies, i.e., *CollectionIteratorAction* and synchronized *ForkAction* behaviors. The *CollectionIteratorAction* has a special semantic different from the other PCM loop entities which cannot be mapped to QPNs when continuous distributions are used for the iteration count. This is due to the fact that the synchronization of two sub-requests generated by a single host request cannot be mapped to QPNs directly, as individual tokens carry no identity and it cannot be decided for two tokens whether or not they belong to the same host request.

The solver tool developed throughout this thesis makes use of the SimQPN simulator [KB06]. The architecture, design and implementation of the tool were presented. The implementation part focuses on QVT Operational, being the primary implementation language. Also the additional mapping to configure the simulator and to measure the metrics, as well as the way the metrics are aggregated, were presented. The main focus of the tool development was to support the evaluation of the PCM-to-QPN mapping.

The evaluation was separated into two parts. The first part evaluated which of the compared solvers support the various PCM features. This resembles a set of integration tests that ensure that the formal mapping works as expected for simple example models. The new PCM-to-QPN mapping covers a much higher number of features than the existing transformation to LQNs.

In the second part of the evaluation, a set of case studies were carried out considering five different PCM models of realistic size and complexity. They were migrated to the employed version of PCM. The workload specifications were updated to ensure that the modeled system is in steady state. This was necessary as the results for mean response times are only meaningful for systems that are not overloaded. In cases where different system usage specifications were available, they were considered in the analysis. For each case study, the resulting PCM instances were analyzed using SimuCom, the new SimQPN solver, LQNS and LQSim. Each simulation run was executed 30 times to obtain statistically sound results.

The results were very encouraging and demonstrated the applicability of the developed transformation in realistic contexts. The new SimQPN solver produced very accurate results (with deviation from the reference values below 15%) in substantially reduced analysis times. In most cases, the overhead could be reduced by an order of magnitude. Compared to the SimuCom reference solver, the new solver showed performance improvements of up to 20 times. LQNS produced equally accurate results for processing resource utilizations, and about 8% less accurate results for throughputs. LQNS exhibited much less accurate results for mean response times, but, being an analytical method, had a much lower analysis overhead. LQSim could only handle one of the case studies and showed similar accuracy to LQNS. The use of an experimental SimQPN extension, allowing to track individual tokens, made it possible to measure response time distributions with the SimQPN solver. A comparison to the SimuCom results showed acceptable results and helped to uncover previously unknown limitations of the new extension. Finally, the case studies helped to guide future development of SimQPN and parts of the PCM-Bench.

The expected benefits of the PCM-to-QPN transformation discussed in Section 1.1 were all achieved. All PCM instances that were available for the thesis could be solved without issues and the SimQPN solver did not run into memory limitations, as SimuCom did. The SimQPN solver was integrated into the PCM-Bench employing the new ProbeSpec meta-model soon to be introduced in SimuCom. The source code has been structured into modules which can be reused for future transformations. The ProbeSpec meta-model allows to customize which metrics are gathered during the simulation without knowing any details about QPNs. Several metrics are offered by the SimQPN solver which are not directly offered by SimuCom or other existing solvers. Notable examples include the utilization of passive resources and the information about how much of the utilization of a processing resource is caused by the individual usage scenarios. The mapping is generic enough to be useful for future transformations to QPNs in the domain of software performance engineering.

It is planned to publish the contributions of this thesis in a paper.

In summary, the primary contributions of this thesis are:

- A formal mapping from PCM to QPNs. Analysis of limitations that apply for each of the mapped features. Analysis of the limitations of the DependencySolver module which is used to resolve PCM stochastic variable dependencies and to assist PCM model traversal.
- Implementation of a PCM-to-QPN transformation in a new PCM solver tool based on SimQPN, a mature simulator for QPNs. The tool successfully reduces the anal-

ysis overhead by at least an order of magnitude while maintaining a high level of results accuracy. The transformation serves as a basis for future transformations to QPNs. Important practical experience with using QVT Operational for a complex transformation was gained. This supports future decisions on model-to-model transformation languages. The tool was also integrated into the PCM-Bench tool, delivered with the PCM meta-model.

- An extensive evaluation of the PCM-to-QPN transformation regarding the results accuracy and analysis overhead. The SimuCom reference solver was compared with the developed SimQPN solver, and LQNS and LQSim, two existing solvers based on LQNs. Customized PCM instances were created for each of the mapped features, showing for the first time in detail, which solver supports which of the multitude of PCM features. Additionally, to evaluate the transformation in realistic conditions, five case studies were conducted using the largest exiting PCM instances that could be obtained for this thesis. One of the case studies was conducted in cooperation with ABB Research, demonstrating the applicability of the results of the thesis in an industrial context.
- Future research areas were identified, especially regarding the PCM stochastic expressions language and the possibilities to reduce expressions to more commonly known probability distributions.

9.2 Future Work

The identified future research areas are grouped according to the three main contributions of this thesis: the mapping, the solver tool implementation and the mapping evaluation. Regarding the mapping, the following tasks have been identified:

- The case study and maintenance work have shown that a deeper analysis of the DependencySolver limitations and generally the handling and reduction of stochastic expressions requires more research. Important research questions remain: Which exact limitations apply when going from value arithmetic of stochastic expressions to an arithmetic involving the folding of the distributions? The SimuCom solver directly uses the stochastic expression to draw a sample value and can therefore evaluate the expression in different contexts. A mapping to a language that does not handle stochastic expressions directly requires to reduce the expressions to one of the supported distributions. It is unclear in which cases this is possible, and which of the possible cases are implemented by the DependencySolver. Another question is how stochastic dependencies between variables can be handled. Currently they are simply ignored by the DependencySolver. This can lead to inaccurate results, e.g., when nested guarded branches use the same stochastic variable in their conditions.
- More research is required with regards to measuring the response time distribution using probes. The two presented loop alternatives, as well as the fork mapping, need to be examined in more detail to determine the degree of error being introduced. The possibility to relax the assumptions in the modeling of loops should be explored.
- SimuCom allows the inclusion of a middleware description model in the simulation process. This possibility is outside the scope of this thesis. Further research is needed to evaluate if those parts could be mapped to QPNs.

The solver tool can be improved in the following ways:

- Once the support for empirical distributions in queueing places of SimQPN has been improved, the transformation of the workload, internal actions, and of linking resources should be updated to use those distributions instead of mean values.

- Once the SimuCom solver and the PCM editors fully support the ProbeSpec metamodel, the corresponding module in the solver tool of this thesis should be updated. The user will then benefit from an improved ProbeSpec user interface and from an integrated way of specifying the metrics at the PCM domain level.
- Once the PCM tools support the new EDP2 data serialization, the corresponding module in the solver tool can be updated to feed the data to EDP2. The user of the SimQPN solver would then gain the benefits of an improved results visualization and possibly of integrated post processing.
- A number of metrics could easily be implemented, further increasing the range of metrics available: throughput per *AbstractUserAction*, breakdown of the metrics per *EntryLevelSystemCall* and per *ExternalCallAction* in addition to per *UsageScenario*. Also, support of response time distributions for all response time metrics (not just *UsageScenario*) would be beneficial.
- The DependencySolver can be extended to support the usage of stochastic variables in QoS annotations.
- The user interface to configure the default ProbeSpec annotation behavior (in case a ProbeSpec model is not specified) should be added.

The following tasks would improve the mapping evaluation:

- The evaluation of the usability, as well as of the portability, of the different solvers is outside the scope of this thesis. A detailed evaluation would be beneficial as they are important factors in the value of the tools for a performance analyst.
- Two mapping variations for loops have been presented. The corresponding results precision and analysis overhead should be evaluated in more detail.
- SimuCom offers a relative precision stopping criterion for the response time of usage scenarios. This requires the measurement of the response times of individual requests. With limitations, this is possible in SimQPN through the probes feature. The feature was added late during the thesis and was not available for the major part of the evaluation. Fixed simulation times were used instead. It would be beneficial to explore the probes feature in combination with a relative precision stopping criterion. There is the potential to remove the assumption that the performance analyst is able to determine a suitable fixed simulation time for a given model.
- Case studies employing more complex stochastic expressions are out of the scope of this thesis. A more detailed analysis of the impact of the numerous simplifications introduced by the DependencySolver and distribution folding would be of value.
- The MediaStore case study appears suitable for a more detailed analysis of the effect of the DependencySolver as all three solvers based on the DependencySolver show about an equal error.

Bibliography

[Bau93] F. Bause, “Queueing Petri Nets – a formalism for the combined qualitative and quantitative analysis of systems,” oct 1993, pp. 14 –23.

[BBK95] F. Bause, P. Buchholz, and P. Kemper, “QPN-tool for the specification and analysis of hierarchically combined queueing petri nets,” in *MMB ’95: Proceedings of the 8th International Conference on Modelling Techniques and Tools for Computer Performance Evaluation.* London, UK: Springer-Verlag, 1995, pp. 224–238.

[BCR94] V. R. Basili, G. Caldiera, and H. D. Rombach, “The Goal Question Metric approach,” in *Encyclopedia of Software Engineering.* Wiley, 1994.

[BDM02] S. Bernardi, S. Donatelli, and J. Merseguer, “From UML sequence diagrams and statecharts to analysable petri net models,” in *WOSP ’02: Proceedings of the 3rd international workshop on Software and performance.* New York, NY, USA: ACM, 2002, pp. 35–45.

[BdWCM05] E. Bondarev, P. de With, M. Chaudron, and J. Muskens, “Modelling of input-parameter dependency for performance predictions of component-based embedded systems,” in *EUROMICRO ’05: Proceedings of the 31st EUROMICRO Conference on Software Engineering and Advanced Applications.* Washington, DC, USA: IEEE Computer Society, 2005, pp. 36–43.

[BGMO06] S. Becker, L. Grunske, R. Mirandola, and S. Overhage, “Performance prediction of component-based systems - a survey from an engineering perspective,” in *Architecting Systems with Trustworthy Components.* Springer Berlin / Heidelberg, 2006, pp. 169–192.

[BK02] F. Bause and P. S. Kritzinger, *Stochastic Petri Nets – An Introduction to the Theory*, 2nd ed. Vieweg Verlag, 2002.

[BKK09] F. Brosig, S. Kounev, and K. Krogmann, “Automated extraction of palladio component models from running enterprise java applications,” in *VALUETOOLS ’09: Proceedings of the Fourth International ICST Conference on Performance Evaluation Methodologies and Tools.* ICST, Brussels, Belgium, Belgium: ICST (Institute for Computer Sciences, Social-Informatics and Telecommunications Engineering), 2009, pp. 1–10.

[BKR09] S. Becker, H. Koziolek, and R. Reussner, “The palladio component model for model-driven performance prediction,” *J. Syst. Softw.*, vol. 82, no. 1, pp. 3–22, 2009.

[BM04] A. Bertolino and R. Mirandola, “CB-SPE tool: Putting component-based performance engineering into practice,” in *Component-Based Software Engineering*, ser. Lecture Notes in Computer Science, I. Crnkovic, J. A. Stafford, H. W. Schmidt, and K. Wallnau, Eds. Springer Berlin / Heidelberg, 2004, vol. 3054, pp. 233–248.

[BMdW$^+$04] E. Bondarev, J. Muskens, P. de With, M. Chaudron, and J. Lukkien, "Predicting real-time properties of component assemblies: a scenario-simulation approach," aug. 2004, pp. 40 – 47.

[CDGDM08] V. Cortellessa, S. Di Gregorio, and A. Di Marco, "Using ATL for transformations in software performance engineering: a step ahead of java-based transformations?" in *WOSP '08: Proceedings of the 7th international workshop on Software and performance.* New York, NY, USA: ACM, 2008, pp. 127–132.

[CH03] K. Czarnecki and S. Helsen, "Classification of model transformation approaches," in *OOPSLA' 03 Workshop on Generative Techniques in the Context of Model-Driven Architecture*, 2003.

[EFH04] E. Eskenazi, A. Fioukov, and D. Hammer, "Performance prediction for component compositions," in *Component-Based Software Engineering*, ser. Lecture Notes in Computer Science, I. Crnkovic, J. A. Stafford, H. W. Schmidt, and K. Wallnau, Eds. Springer Berlin / Heidelberg, 2004, vol. 3054, pp. 280–293.

[emf] Eclipse Modeling Framework Project (EMF). `http://www.eclipse.org/modeling/emf/`.

[GM01] H. Gomaa and D. Menascé, "Performance engineering of component-based distributed software systems," in *Performance Engineering*, ser. Lecture Notes in Computer Science, R. Dumke, C. Rautenstrauch, A. Scholz, and A. Schmietendorf, Eds. Springer Berlin / Heidelberg, 2001, vol. 2047, pp. 40–55.

[GMS07] V. Grassi, R. Mirandola, and A. Sabetta, "Filling the gap between design and performance/reliability models of component-based systems: A model-driven approach," *J. Syst. Softw.*, vol. 80, no. 4, pp. 528–558, 2007.

[GP02] G. P. Gu and D. C. Petriu, "XSLT transformation from UML models to LQN performance models," in *WOSP '02: Proceedings of the 3rd international workshop on Software and performance.* New York, NY, USA: ACM, 2002, pp. 227–234.

[Hei07] F. Heimburger, "Performance Modelling of Java EE Applications using LQNs and QPNs," Master's thesis, TU Darmstadt, 2007.

[Hen10] J. Henss, "Performance prediction for highly distributed systems," in *Proceedings of the Fifteenth International Workshop on Component-Oriented Programming (WCOP) 2010*, ser. Interne Berichte, B. Bühnová, R. H. Reussner, C. Szyperski, and W. Weck, Eds., vol. 2010-14. Karlsruhe, Germany: Karlsruhe Institue of Technology, Faculty of Informatics, June 2010, pp. 39–46.

[HKW$^+$08] S. Herold, H. Klus, Y. Welsch, C. Deiters, A. Rausch, R. Reussner, K. Krogmann, H. Koziolek, R. Mirandola, B. Hummel, M. Meisinger, and C. Pfaller, *The Common Component Modeling Example*, ser. Lecture Notes in Computer Science. Springer-Verlag Berlin Heidelberg, 2008, vol. 5153, ch. CoCoME – The Common Component Modeling Example, pp. 16–53.

[HMSW02] S. Hissam, G. Moreno, J. Stafford, and K. Wallnau, "Packaging predictable assembly," in *Component Deployment*, ser. Lecture Notes in Computer Science, J. Bishop, Ed. Springer Berlin / Heidelberg, 2002, vol. 2370, pp. 108–124.

[JK06a] F. Jouault and I. Kurtev, "Transforming models with ATL," in *Satellite Events at the MoDELS 2005 Conference*, ser. Lecture Notes in Computer Science, vol. 3844. Berlin: Springer Verlag, 2006, pp. 128–138.

[JK06b] ——, “On the architectural alignment of ATL and QVT,” in *SAC ’06: Proceedings of the 2006 ACM symposium on Applied computing.* New York, NY, USA: ACM, 2006, pp. 1188–1195.

[KB03] S. Kounev and A. Buchmann, “Performance modelling of distributed e-business applications using queuing petri nets,” in *ISPASS ’03: Proceedings of the 2003 IEEE International Symposium on Performance Analysis of Systems and Software.* Washington, DC, USA: IEEE Computer Society, 2003, pp. 143–155.

[KB06] ——, “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, 2006.

[KBHR10] S. Kounev, F. Brosig, N. Huber, and R. Reussner, “Towards self-aware performance and resource management in modern service-oriented systems,” in *Proceedings of the 7th IEEE International Conference on Services Computing (SCC 2010), July 5-10, Miami, Florida, USA.* IEEE Computer Society, 2010.

[KD09] S. Kounev and C. Dutz, “QPME - A Performance Modeling Tool Based on Queueing Petri Nets,” *ACM SIGMETRICS Performance Evaluation Review (PER), Special Issue on Tools for Computer Performance Modeling and Reliability Analysis*, vol. 36, no. 4, pp. 46–51, March 2009.

[KKC00] R. Kazman, M. Klein, and P. Clements, “ATAM: Method for architecture evaluation,” CMU/SEI, Tech. Rep., 2000.

[Kou06] S. Kounev, “Performance modeling and evaluation of distributed component-based systems using queueing petri nets,” *IEEE Trans. Softw. Eng.*, vol. 32, no. 7, pp. 486–502, 2006.

[Koz08] H. Koziolek, “Parameter dependencies for reusable performance specifications of software components,” Ph.D. dissertation, Fakultät für Informatik (Fak. f. Informatik) Institut für Programmstrukturen und Datenorganisation (IPD), 2008.

[Koz09] ——, “Performance evaluation of component-based software systems: A survey,” *Performance Evaluation*, vol. In Press, Corrected Proof, pp. –, 2009.

[KR08] H. Koziolek and R. Reussner, “A model transformation from the palladio component model to layered queueing networks,” in *SIPEW ’08: Proceedings of the SPEC international workshop on Performance Evaluation.* Berlin, Heidelberg: Springer-Verlag, 2008, pp. 58–78.

[KWB03] A. G. Kleppe, J. Warmer, and W. Bast, *MDA Explained: The Model Driven Architecture: Practice and Promise.* Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 2003.

[MB04] M. Marzolla and S. Balsamo, “UML-PSI: The UML performance simulator,” *Quantitative Evaluation of Systems, International Conference on*, vol. 0, pp. 340–341, 2004.

[MI04] A. D. Marco and P. Inverardi, “Compositional generation of software architecture performance QN models,” *Software Architecture, Working IEEE/IFIP Conference on*, vol. 0, p. 37, 2004.

[MKBR10] A. Martens, H. Koziolek, S. Becker, and R. H. Reussner, “Automatically improve software models for performance, reliability and cost using genetic algorithms,” in *WOSP/SIPEW ’10: Proceedings of the first joint WOSP/SIPEW international conference on Performance engineering.* New York, NY, USA: ACM, 2010, pp. 105–116.

[MM06] A. D. Marco and R. Mirandola, "Model transformation in software performance engineering," in *QoSA*, 2006, pp. 95–110.

[OMG05] *UML Profile For Schedulability, Performance, And Time, Version 1.1*, http://www.omg.org/technology/documents/formal/schedulability.htm, OMG - Object Management Group, Inc. Std., January 2005.

[OMG06a] *Meta Object Facility (MOF) Core Specification, Version 2.0*, http://www.omg.org/spec/MOF/2.0, OMG - Object Management Group, Inc. Std., January 2006.

[OMG06b] *Object Constraint Language (OCL), Version 2.0*, http://www.omg.org/spec/OCL/2.0, OMG - Object Management Group, Inc. Std., May 2006.

[OMG07] *Unified Modeling Language (UML) Specification, Version 2.1.2*, http://www.omg.org/spec/UML/2.1.2, OMG - Object Management Group, Inc. Std., November 2007.

[OMG08a] *Meta Object Facility (MOF) 2.0 Query/View/Transformation Specification, Version 1.0*, http://www.omg.org/spec/QVT/1.0, OMG - Object Management Group, Inc. Std., April 2008.

[OMG08b] *UML Profile for Modeling and Analysis of Real-Time and Embedded Systems (MARTE), Beta 2*, http://www.omgmarte.org/Specification.htm, OMG - Object Management Group, Inc. Std., June 2008.

[PW07] D. Petriu and M. Woodside, "An intermediate metamodel with scenarios and resources for generating performance models from UML designs," *Software and Systems Modeling (SoSyM)*, vol. 6, no. 2, pp. 163–184, June 2007.

[qim] Q-ImPrESS project results. http://www.q-impress.eu/wordpress/publications/.

[RH08] R. Reussner and W. Hasselbring, *Handbuch der Software-Architektur*, 2nd ed. Heidelberg: dpunkt, 2008.

[SKK$^+$01] M. Sitaraman, G. Kulczycki, J. Krone, W. F. Ogden, and A. L. N. Reddy, "Performance specification of software components," *SIGSOFT Softw. Eng. Notes*, vol. 26, no. 3, pp. 3–10, 2001.

[SLC$^+$05] C. U. Smith, C. M. Lladó, V. Cortellessa, A. D. Marco, and L. G. Williams, "From UML models to software performance results: an SPE process based on XML interchange formats," in *WOSP '05: Proceedings of the 5th international workshop on Software and performance.* New York, NY, USA: ACM, 2005, pp. 87–98.

[Sta73] H. Stachowiak, *Allgemeine Modelltheorie.* Wien - New York: Springer-Verlag, 1973.

[SV06] T. Stahl and M. Völter, *Model Driven Software Development: Technology, Engineering, Management.* John Wiley & Sons, 2006.

[TG08] M. Tribastone and S. Gilmore, "Automatic extraction of PEPA performance models from UML activity diagrams annotated with the MARTE profile," in *WOSP '08: Proceedings of the 7th international workshop on Software and performance.* New York, NY, USA: ACM, 2008, pp. 67–78.

[Tri10] M. Tribastone, “Relating layered queueing networks and process algebra models,” in *WOSP/SIPEW ’10: Proceedings of the first joint WOSP/SIPEW international conference on Performance engineering.* New York, NY, USA: ACM, 2010, pp. 183–194.

[WW04] X. Wu and M. Woodside, “Performance modeling from software components,” *SIGSOFT Softw. Eng. Notes*, vol. 29, no. 1, pp. 290–301, 2004.