Automated Transformation of Component-based Software Architecture Models to Queueing Petri Nets

Philipp Meier
Karlsruhe Institute of Technology (KIT)
76131 Karlsruhe, Germany
mail@philippmeier.com

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

Heiko Koziolek
Industrial Software Systems
ABB Corporate Research
68526 Ladenburg, Germany
heiko.koziolek@de.abb.com

***Abstract*—Performance predictions early in the software development process can help to detect problems before resources have been spent on implementation. The Palladio Component Model (PCM) is an example of a mature domain-specific modeling language for component-based systems enabling performance predictions at design time. PCM provides several alternative model solution methods based on analytical and simulation techniques. However, existing solution methods suffer from scalability issues and provide limited flexibility in trading-off between results accuracy and analysis overhead. Queueing Petri Nets (QPNs) are a general-purpose modeling formalism, at a lower level of abstraction, for which efficient and mature simulation-based solution techniques are available. This paper contributes a formal mapping from PCM to QPN models, implemented by means of an automated model-to-model transformation as part of a new PCM solution method based on simulation of QPNs. The limitations of the mapping and the accuracy and overhead of the new solution method compared to existing methods are evaluated in detail in the context of five case studies of different size and complexity. The new solution method proved to provide good accuracy with solution overhead up to 20 times lower compared to PCM's reference solver.**

I. Introduction

One of the most important extra-functional properties of modern software systems is their performance. Architectural changes in the late development stages are very costly. Therefore, it is essential to be able to predict the system performance at design time in order to detect potential problems before resources have been spent on implementation. The Palladio Component Model (PCM) [1] is a domain-specific modeling language for component-based systems enabling performance predictions at design time. Four performance-influencing factors are modeled for each software component: the component implementations, the external services they use, the execution environment on which they are deployed, and the component usage profiles.

PCM models are analyzed through a transformation to a predictive performance model at a lower level of abstraction. In the case of SimuCom, the solver distributed with the PCM meta-model, the transformation targets Java sourcecode based on a general-purpose simulation framework. Even though SimuCom uses a transformation, it does not abstract away information from the model and implements the full model semantics. SimuCom was chosen as the reference solver for the evaluation as it is the most mature method, having been used and improved since the first version of PCM. Other existing transformations are a transformation to Layered Queueing Networks (LQNs) [2] and a transformation to Stochastic Regular Expressions [3]. They use simplifications and abstractions to reduce the evaluation overhead. 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 either provide high precision at a high overhead, or low precision at a low overhead and they also suffer from scalability issues.

Queueing Petri Nets (QPNs) are a general-purpose modeling formalism, at a lower level of abstraction compared to PCM, that has been shown to lend itself well to modeling and analyzing the performance of distributed component-based systems [4], [5]. A mature and optimized simulation engine (SimQPN [6] which is part of QPME – the Queueing Petri net Modeling Environment [7]), as well as analytical techniques (e.g., HiQPN-Tool [8]), are available. However, QPNs are a general-purpose modeling formalism and therefore provide no constructs for representing software domain elements like components or system usage profiles directly. A mapping from the components of the system to the appropriate QPN model currently has to be developed manually and individually for each project. It is therefore desirable to be able to model the system in PCM, which supports most system entities directly, and conduct the analysis using the available tools and methods for QPNs, in particular the highly optimized SimQPN simulator [6]. An automatic transformation from PCM to QPN models would open up the benefits of QPNs to the PCM community and provide a basis for future transformations to QPNs from other source models in the performance engineering domain.

This paper makes the following 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, iii) an extensive evaluation of the PCM-to-QPN transformation in terms of results accuracy and analysis overhead. While the transformation is defined in the context of PCM, it is not limited to PCM and can be easily generalized to other component-based software architecture models. Further details on the presented results, including a more detailed description of the mapping and a more in-depth comparison of the different solvers, can be found in [9].

The remainder of this paper is organized as follows: Section II discusses the related work of our approach. Section III briefly introduces the foundations of this work: PCM and QPNs. The mapping from PCM to QPNs is presented in Section IV, its implementation in Section V. Section VI presents the evaluation of the approach, including the results of the case studies. Finally, Section VII concludes this paper and discusses future work.

II. Related Work

Performance prediction approaches for component-based systems have been surveyed by Koziolek [10]. Transformations in the software performance engineering domain are discussed in [11]. This section mainly covers other PCM-transformations as UML-based transformations and transformations for proprietary languages are only remotely related.

Existing solvers for PCM based on model-to-model transformations are directly related to this work. Two solvers, built on top of the DependencySolver introduced in [3], are based on a transformation to Layered Queueing Networks (LQNs) [2] and a transformation to Stochastic Regular Expressions [3]. The DependencySolver is a module that automatically resolves parametric dependencies and stochastic expressions. Similarly to the above two transformations, we use it as a preprocessing step in the PCM-to-QPN transformation presented in this paper. A detailed comparison of LQNs and QPNs as analysis formalisms can be found in [12]. Stochastic Regular Expressions can be solved analytically with very low overhead, however, they only support single user scenarios. Koziolek [3] introduces a transformation to an extended form of QPNs which has not been studied by the research community and is not supported by available tools. It uses tokens that carry arbitrary properties instead of just a color value. Henss [13] proposes a PCM transformation to OMNeT++, focusing on realistic network infrastructure closer to the OSI reference network model. No experimental evaluation of the approach has been published. The PCM-Bench tool comes with the SimuCom simulator [1]. A model-to-text transformation is used to generate Java code 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. However, the current version of SimuCom is limited in terms of the supported metrics and provides limited configurability of the simulation output data. Furthermore, it suffers from scalability issues, due to the high analysis overhead and memory constraints.

Remotely relevant to this paper are other model-based approaches that use some form of transformation to analyze model instances. LQNs are the most common analysis technique, but Markov chains, Stochastic Petri Nets and Stochastic Process Algebras are also used for performance analysis [1]. Many approaches are based on the Unified Modeling Language (UML) [14], e.g., [15], [16], [17], [18], often extended with an annotation profile like SPT (UML Profile for Schedulability, Performance and Time) [19], or the newer and extended version MARTE (UML Profile for Modeling and Analysis of Real-time and Embedded Systems) [20]. Examples include [21], [22], [11], [23], [24]. The main difference to our approach is that the use of UML implies customary meta-models for specifying the performance aspects of a system, that often significantly differ from PCM. A detailed discussion of the benefits and drawbacks of PCM compared to UML is presented in [1].

III. Foundations

We provide an overview of the Palladio Component Model and Queuing Petri Nets, which form the basis of this work.

A. Palladio Component Model (PCM)

The Palladio Component Model (PCM) is a meta-model allowing the specification of performance-relevant information of a component-based architecture [1]. 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: its implementation, the performance of external services it requires, the performance of the execution environment it is deployed on, and the usage profile. These aspects are specified using parametric dependencies, providing a domain specific language for each of the four roles in the CBSE development process: component developer, software architect, system deployer and business domain expert. Each role contributes a part of the PCM instance to be analyzed.

The *component developer* specifies the implementation-specific information of a component and stores it in 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 model the externally visible behavior of a service with resource demands and calls to required services.

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 provided and required 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.

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 occupy the resources of the resource container the assembly context is deployed on.

The *domain expert* specifies 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 accurately represent real life scenarios.

Many PCM entities contain *RandomVariable*s representing arbitrary discrete and continuous statistical distributions. They are specified using the PCM stochastic expressions language (StoEx). 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. A detailed examination and a description of the underlying meta-model can be found in [3].

The DependencySolver [3] 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.

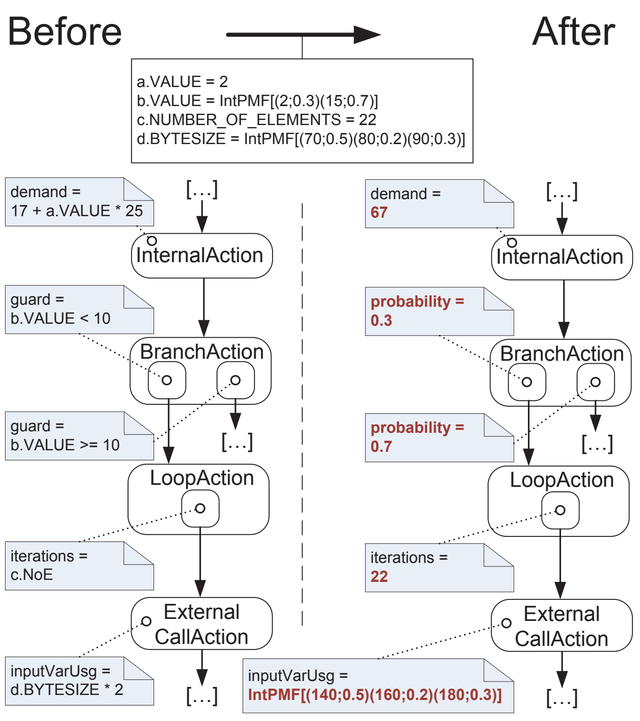


Fig. 1. DependencySolver Illustration [3]

Figure 1 illustrates the basic idea behind the DependencySolver. The example shows a simplified SEFF with parametric dependencies in the form of stochastic expressions. An *IntPMF* is a probabilistic mass function with integral values. The values for the employed variables (e.g., *a.VALUE*), set in the usage profile, are used by the DependencySolver to reduce the expressions to concrete representations. Computations within the expressions are also carried out.

The context shown in the example is simplified, as the actual context is only known at analysis time and depends on the different PCM sub-models (e.g., repository and system assembly). One SEFF is usually traversed with a range of different contexts, resulting in different analysis-time instances. The main simplification introduced by the DependencySolver is that all information about stochastic variables is lost, and with it the variable scopes.

B. Queueing Petri Nets (QPNs)

Queueing Petri Nets (QPNs) build on Colored Generalized Stochastic Petri Nets (CGSPNs), a combination of Colored Petri Nets (CPNs) and Generalized Stochastic Petri Nets (GSPNs) [25]. A formal definition can be found in [26]. 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 the number of tokens a transition requires in each connected input place to be ready to fire. When a transition fires, it consumes the listed number tokens from the incoming places and deposits an independent number of tokens in connected output places according to a *backward incidence function*. If no target places are listed, the tokens are effectively removed from the net. If more than one transition is ready to fire, one is randomly chosen with equal probability.

CPNs introduce the ability to distinguish different token classes using *token colors*. The initial marking and incidence functions are now defined with respect to multiple different colors. The different possibilities of firing a transition are referred to as *modes*. Modes also have firing weights that influence which mode is chosen when multiple modes are ready to fire. GSPNs introduce timed transitions and firing weights for transitions, both of which are not used by our mapping. *Timed transitions* place restrictions on when the transition can be fired, even when all necessary input tokens are available. The original transitions do not have timing restrictions and are therefore called immediate transitions. QPNs introduce a new type of place: the *queueing place*. A queueing place consists of a queue, and a depository. Tokens are placed inside the queue according to a certain scheduling strategy. The time it takes to service a token is defined through a statistical distribution. Once a token has completed its service, it is put into the depository, which then behaves like a regular (ordinary Petri net) place for connected transitions. Only tokens in the depository are considered available for the incidence functions. Furthermore, QPNs can be nested using *subnet places*, that contain an arbitrary subnet with a single entry and exit place.

QPNs were chosen of other variants of Petri nets, because queuing places are well-suited to model the hardware contention of a system, while, at the same time, places and transitions allow for an accurate model of software contention. LQN has similar properties, but we deliberately wanted to explore the suitability of QPNs compared to LQNs in regard to performance predictions.

IV. PCM to QPN Mapping

This section presents the main contribution of this paper: the mapping from PCM models to QPNs. The limitations applying to the individual features are also discussed. All features of PCM in version 3.2 are covered. They are presented groups, each having its own subsection: *workload*, *calls*, *branches*, *loops*, *forks*, *processing resources* and *passive resources*. The mapping of *linking resources* and the mapping of *QoS annotations*, as well as detailed PCM meta-model excerpts, have been omitted for reasons of brevity and can be found in [9]. The QPN diagrams follow the notation presented in Figure 2. A *Subnet* represents a part of the QPN which varies independently of the described feature. All transitions used are immediate transitions.

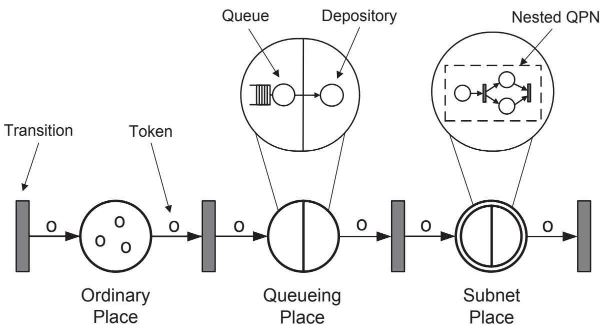
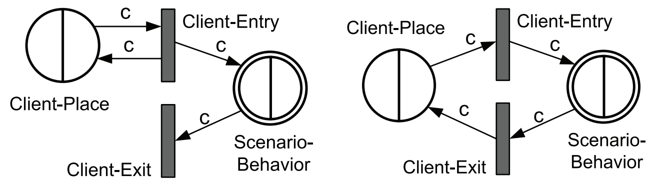


Fig. 2. QPN Diagram Notation

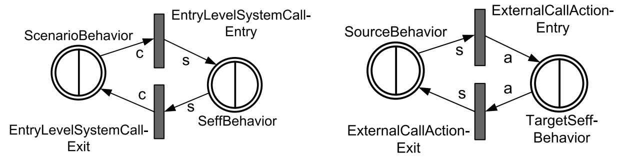
A. Workload

The workload in PCM is represented as a number of usage scenarios, running in parallel. Each scenario has a workload specification and a *ScenarioBehavior*. The *ScenarioBehavior* is very similar to the SEFF of a component, but does not consume resources directly. The calls reference external inferfaces of the system assembly. PCM supports both open and closed workloads [27]. An open workload is characterized by an inter-arrival time distribution, which describes the time that elapses between consecutive requests. Figure 3(a) shows how the *OpenWorkload* usage model entity is represented in the generated QPN to achieve the open workload semantics. The *Client-Place* queueing place generates tokens of a color c which is different for each *UsageScenario*. It references a client queue with Infinite Server (IS) scheduling strategy. An empirical distribution is used for the *Client-Place* resource demand representing *inter-arrival time* distribution of the



(a) OpenWorkload (b) ClosedWorkload

Fig. 3. Workload QPN



(a) EntryLevelSystemCall (b) ExternalCallAction

Fig. 4. Calls QPN

OpenWorkload. The initial number of tokens is set to 1. For each input token the *Client-Entry* transition creates a new token in the subnet representing the *ScenarioBehavior*. Another token is created in the *Client-Place* queue. The *Client-Exit* transition destroys tokens of color c from the *ScenarioBehavior* subnet.

A closed workload is characterized both by an integral population, as well as by a think time distribution. There is a fixed number of requests, each of which has to wait according to the think time after its processing is complete. Figure 3(b) shows the mapping for a *ClosedWorkload*. The 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 the *population* of the closed workload. A new token will now be available after it has passed through the whole *ScenarioBehavior* and after the residence time in the queue, which equals the *think time*.

B. Calls

In PCM 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. The difference is that *EntryLevelSystemCall*s are part of the *UsageModel* and reference only external interfaces of the system assembly. Figure 4(a) and Figure 4(b) show the mapping. Each call entity is represented by two transitions.

EntryLevelSystemCall-Entry connects the subnet of the previous action in the current usage model behavior with the subnet of the target SEFF . *EntryLevelSystemCall-Exit* in return connects the target SEFF subnet with the subnet of the following action. A new token color is created for each *EntryLevelSystemCall* that distinguishes the requests coming from that call.

The mapping of *ExternalCallAction* is defined in a similar way. The new token color is derived from the current token number during traversal.

C. 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 branch entities of PCM (*Branch* and *BranchAction*) are mapped to the same basic QPN elements shown in Figure 5. The transitions leading back to the source behavior are trivial and have been omitted. The incidence functions are described in the following.

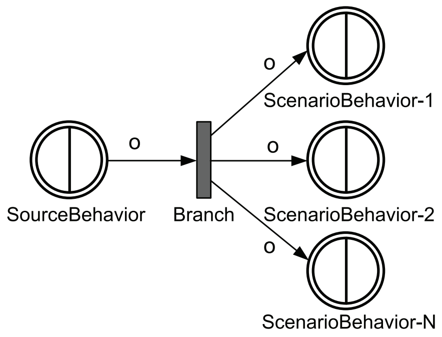


Fig. 5. Branch QPN

A branch has N child behaviors B_i with corresponding branching probabilities P_i. The *Branch* transition consumes a token from the subnet of its *predecessor* (forward incidence function). Its backward incidence function has N modes corresponding to the number of child behaviors. Each mode creates a token in the subnet for child behavior B_i and has a firing weight of the branching probability P_i.

D. Loops

PCM supports three different types of loops: *Loop* in usage models, and *LoopAction* and *CollectionIteratorAction* in SEFFs. Using the DependencySolver, all three variants are reduced to the following behavior specification:

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.

Two different QPN mapping variants were developed. The first variant, shown in Figure 6 was employed for the case studies presented later as it uses fewer places, modes and tokens. Figure 7 shows the alternative variant. As the two alternatives share considerable parts, the common parts will be discussed first.

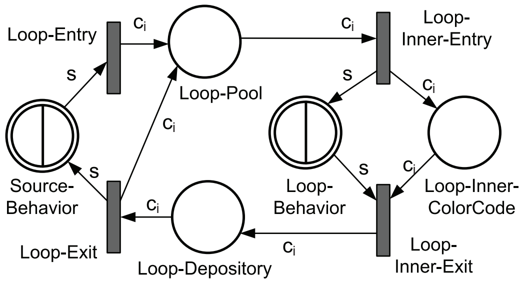


Fig. 6. Loop QPN

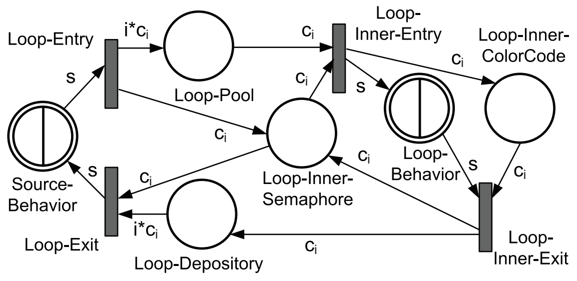


Fig. 7. Loop QPN (Alternative)

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* subnet and token output to the *successor* subnet. The inner part handles the input and output to and from the child behavior subnet, denoted as *LoopBehavior*. The color of the input and output 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 behavior property 2) above. If only one token color was used, the exit transition would not know when the whole request is 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 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 behavior property 3) is implemented. The first approach, in Figure 6, 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, which can be found in [9], does not impose this simplification. We skip its presentation here for the sake of compactness.

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. In QPNs, transition firing probabilities are always evaluated independently of other transitions. Any dependencies would have to be encoded in colors. As only a fixed number of colors is available, dependencies between continuous variables cannot be mapped adequately. An extreme number of colors is also inefficient.

E. Forks

PCM supports both synchronous and asynchronous forks. Both are realized by a single entity, the *ForkAction*. It directly contains a number of N asynchronous *ForkedBehavior*s. In addition, it contains a *SynchronizationPoint* which contains M synchronized *ForkedBehavior*s.

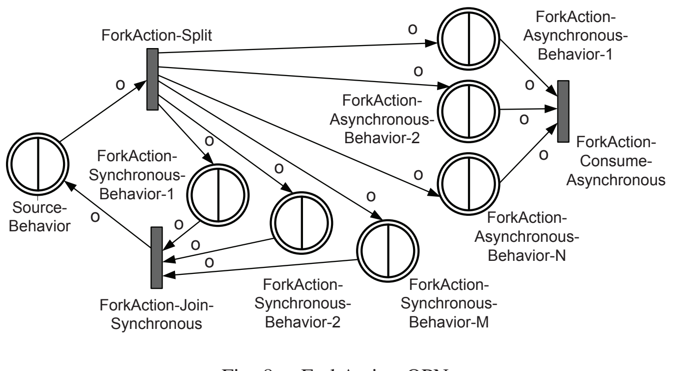

ForkAction-Split
ForkAction-Asynchronous-Behavior-1
ForkAction-Asynchronous-Behavior-2
ForkAction-Consume-Asynchronous
ForkAction-Asynchronous-Behavior-N
Source-Behavior
ForkAction-Synchronous-Behavior-1
ForkAction-Join-Synchronous
ForkAction-Synchronous-Behavior-2
ForkAction-Synchronous-Behavior-M
o

Fig. 8. ForkAction QPN

Figure 8 shows the QPN mapping for a *ForkAction*. There are three transitions. *ForkAction-Split* consumes a token from the *predecessor* subnet and creates a token in each of the $M + N$ child behavior subnets. *ForkAction-Consume-Asynchronous* consumes tokens from the N asynchronous child behavior subnets, but does not create any new tokens. *ForkAction-Join-Synchronous* waits until a token is available in each of the M synchronous child behavior subnets. It then consumes all of them and creates a new token in the *successor* subnet. When $M = 0$, a dummy ordinary place is created for the synchronized client behavior subnet to prevent the request from getting lost.

A limitation applies to the mapping of synchronized forked behavior. The synchronization of two sub-requests generated by a single parent request cannot be represented in a semantically equivalent fashion using QPN constructs. Individual tokens carry no identity and it cannot be decided for two tokens whether or not they belong to the same parent request. The tokens are consumed without considering their parent 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.

F. Processing Resources

The PCM version employed for this paper supports only single-server processing resources and three scheduling strategies: first-come-first-served, processor-sharing and infinite server. Each *ResourceContainer* of the *ResourceEnvironment* can have an arbitrary number of *ProcessingResourceSpecification*s which each have a locally unique name, a locally unique *ProcessingResourceType* and a *processingRate*. An *InternalAction* represents a load a request places on one or more processing resources. It contains a number of N *ResourceDemand*s. Each *ResourceDemand* contains a resource demand *specification* in the form of a *PCMRandomVariable* and a reference to a *ProcessingResourceType*. The *specification* uses abstract units of time and must logically match the processing rate of the target *ProcessingResource*. The actual resource demand times are determined at model analysis time using the current *AssemblyContext*.

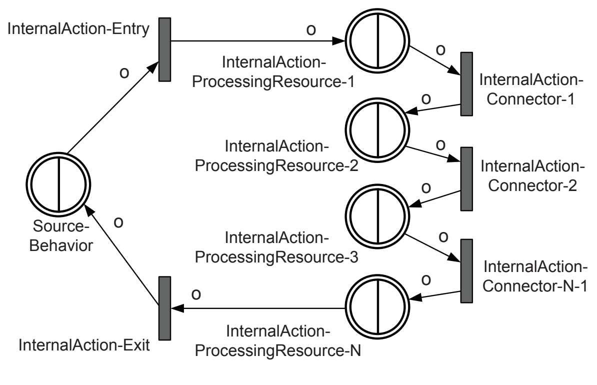

InternalAction-Entry
InternalAction-ProcessingResource-1
InternalAction-Connector-1
InternalAction-ProcessingResource-2
InternalAction-Connector-2
Source-Behavior
InternalAction-ProcessingResource-3
InternalAction-Connector-N-1
InternalAction-Exit
InternalAction-ProcessingResource-N
o

Fig. 9. InternalAction QPN

Figure 9 shows the QPN mapping for an *InternalAction*. The transitions *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 simulated by the SimuCom reference simulator. For each of the N *ResourceDemand*s 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 solves 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.

G. Passive Resources

PCM supports passive resources. A *BasicComponent* can contain N *PassiveResource*s. Each *PassiveResource* contains an initial *capacity* specification of type integer. While *PassiveResource*s are defined per component, they are instantiated per *AssemblyContext*. *AcquireAction* and *ReleaseAction* mark the section during which a request requires a passive resource. They both reference one of the N *PassiveResource*s.

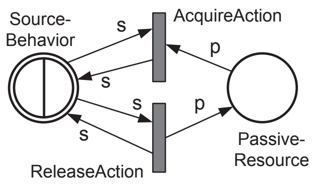


Fig. 10. AcquireAction and ReleaseAction QPN

Figure 10 shows the QPN mapping for both *AcquireAction*s and *ReleaseAction*s. The *PassiveResource* is mapped to an ordinary place. A global semaphore color type is used. The initial number of tokens is set to the *capacity*. The *AcquireAction* transition consumes a token from the *predecessor* subnet and one token from the *PassiveResource* place and generates one token in the *successor* subnet. Similarly, the *ReleaseAction* transition consumes a token from the *predecessor* subnet and creates a token in both the *successor* subnet and the *PassiveResource* place. For each *AssemblyContext* referencing the parent *BasicComponent* of the *PassiveResource*, a different place is created and accessed. For simplicity, this has been omitted from the figure.

V. Implementation

The presented PCM-to-QPN transformation was implemented using QVTO Operational [28] and Java. The transformation itself is largely generic and could easily be reused for a number of solution techniques for QPNs. In this case, we used the SimQPN simulator, which is available as an Eclipse plugin. The tool implementing our new SimQPN-based solver is also available as an Eclipse plugin. It is integrated into the PCM-Bench, which is distributed with the PCM metamodel. Figure 11 outlines the architecture of the tool. The *Instrumentation UI* and *Instrumentation Model* allow the user to specify which metrics to collect at the PCM domain level. Likewise, the *Results Integration* module converts SimQPN-level simulation results back into the PCM domain.

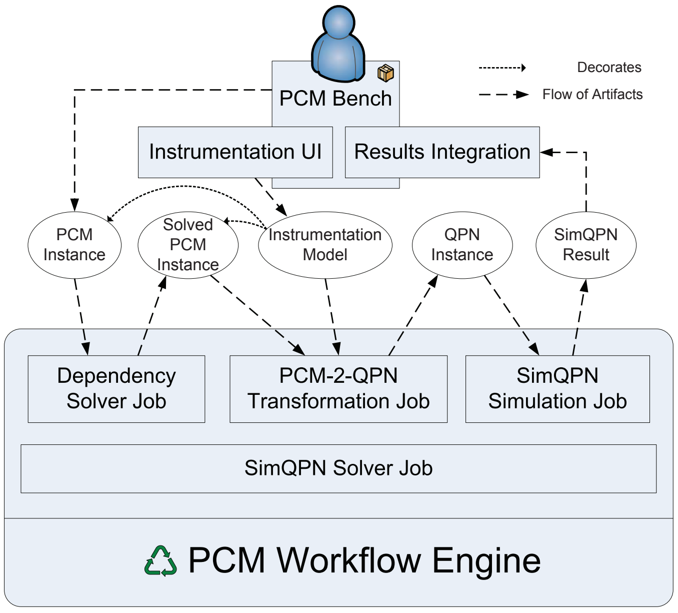


Fig. 11. SimQPN Solver Architecture

VI. Evaluation

This section presents a detailed evaluation of the proposed PCM to QPN mapping. Five case studies were conducted to evaluate the results accuracy and analysis overhead of the new SimQPN Solver in a realistic context. The SimQPN Solver results were compared to the results of the SimuCom reference simulator as well as to the LQNS and LQSim solvers.

In the following, the experimental environment is presented. The individual case studies and their complexity are discussed next. Finally, after outlining the steps taken for each case study, the analysis results are presented. For lack of space, we only present a summary of the evaluation results here. The detailed results are available in [9] including a detailed evaluation of the mapping on a feature-by-feature basis.

A. Experimental Environment

All experiments were conducted 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. The system and hardware configuration included: 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. LQNS and LQSim were available in version 4.1.

Of the available simulation methods offered by SimQPN, the batch means method [29] was used for all simulation runs. It is the most stable method provided by SimQPN. LQSim was used with the `-T<logical runtime>` option. For LQNS the default settings of the PCM-to-LQN transformation were used: *convValue = 1e-005*, *itLimit = 50*, *printInt = 10*, *underCoeff = 0.5* and *psQuantum = 0.001*.

B. Case Studies

The aim of the evaluation was to use PCM models of realistic size and complexity. Instead of creating an artificial model, five case studies were conducted using existing models from various sources: SPECjEnterprise2010, ABB Demonstrator, MediaStore, CoCoME and Business Reporting System.

The SPECjEnterprise2010 model is taken from [30] where it was used to evaluate a method for automated extraction of PCM model instances from running enterprise Java applications. SPECjEnterprise2010 is a benchmark developed by the Standard Performance Evaluation Corporation (SPEC). Two different models were available that differ in their usage model complexity.

In the context of an internship at ABB Research in Ladenburg, Germany, the ABB Demonstrator model was made available, which represents a large distributed factory process control system by ABB [31, D7.1].

The MediaStore is an example model that does not reach the complexity of an industrial system but is still much larger than the simple models used to evaluate individual features in [9]. The PCM instance used for this paper is an adaptation of the MediaStore model used in [3] ported to the current PCM version.

CoCoME stands for Common Component Modeling Example and describes a trading system as it can be observed in a supermarket handling sales [32].

The Business Reporting System (BRS) model is taken from [33]. It was created to evaluate a method of automatic software architecture optimization.

For the SPECjEnterprise2010 and the ABB Demonstrator case studies, the workload parameters were varied, creating an additional 12 model instances in addition to the six models discussed above.

Table I shows a number of complexity metrics that characterize the different case study models. Only components and SEFFs directly or indirectly referenced from the usage model are counted. The number of calls, branches and loops refer to the number of entities that are instantiated during the model traversal of both usage model and system assembly. This better reflects the different combinations of input parameters. The StoEx usage shows the most advanced elements in the employed stochastic expressions.

To further illustrate the complexity of the involved transformations, Figure 12 shows a simplified version of the MediaStore model and the corresponding generated QPN. The incidence functions are omitted to save space. MediaStore is the simplest of the case studies, the QPNs generated from the other case studies are much larger. As an example, the generated QPN model for the BRS case study included a total of 236 places, 171 transitions and 105 token colors.

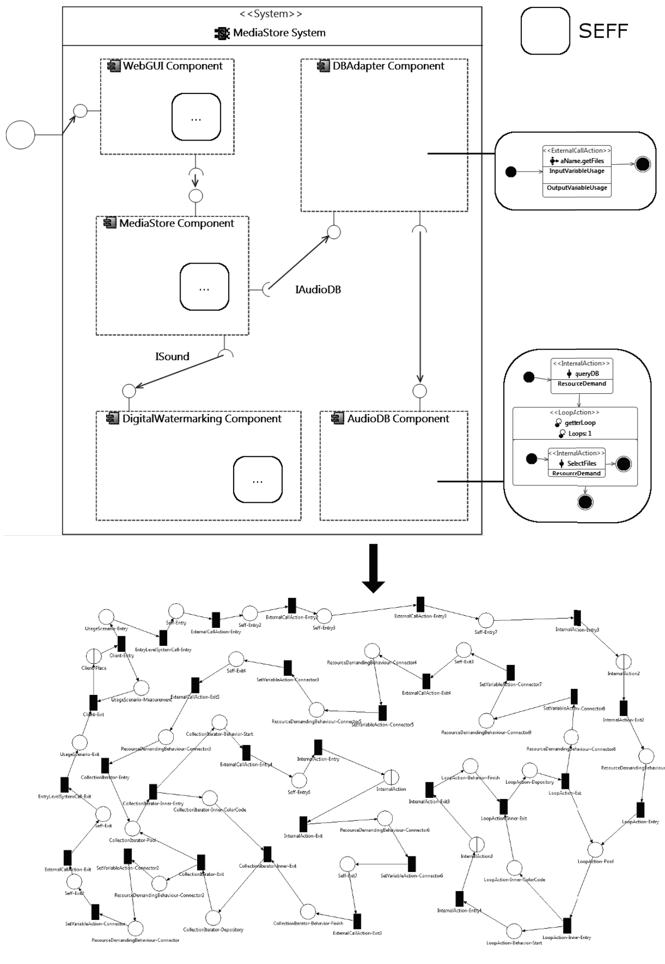


Fig. 12. Example transformation for the MediaStore case study (PCM and QPN models simplified for compactness)

C. Evaluation Steps

To conduct the case studies, an appropriate logical simulation length was determined for each of the considered workload scenarios. A fixed simulation time was employed to maximize the comparability of the results in terms of simulation overhead. All employed simulation based solvers (SimuCom, SimQPN and LQSim) support a fixed simulation time mode of operation. LQNS was set up to use a 95% confidence level.

The evaluation is then based on the total analysis time, which includes the simulation time, as well as any time needed for the actual transformation or related processing.

Using the determined simulation time, each workload scenario was evaluated 30 times with each solver. Then, for

TABLE I
MODEL COMPLEXITY

Model	#UsageScenarios (#Open / #Closed)	#Components (referenced)	#SEFFs (referenced)	#Calls (instantiated)	#Loops (instantiated)	#Branches (instantiated)	StoEx usage
SPECjE (A)	1 / 0	7	7	7	1	1	Exp in workload, IntPMF
SPECjE (B)	1 / 0	7	15	26	2	4	“
ABB	4 / 0	9	9	76	0	76	basic number arithmetic
MediaStore	0 / 1	5	5	5	2	0	IntPMF, DoublePDF
CoCoME	0 / 1	10	12	15	6	3	IntPMF, DoublePDF
BRS	1 / 0	9	27	85	18	20	EnumPDF, DoublePDF

each metric, the relative deviation of the mean value provided by each solver compared to the SimuCom reference solver was computed. To determine, if that deviation is statistically significant, a 95% confidence interval for the difference of means between the SimuCom result and each other solver was computed.

D. Evaluation Results

Table II presents the results accuracy of the most important mean value metrics which were available for each of the case studies. 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. For each metric, the average relative difference of the results for each of the considered solvers compared to the results of the SimuCom reference solver are shown. Between the different workloads of a case study, only the worst case results are presented. If the difference between the solver mean and the reference mean was not statistically significant at a 95% confidence level, $\approx 0\%$ is shown. Where no results are included, no meaningful results could be provided by the respective solver. Table III shows the average total analysis runtimes in seconds. The time needed for the transformation is negligible for models of realistic sizes and is not separated from the dominating simulation time.

In this paper, the results for a single logical simulation runtime chosen for each case study are presented. The results for two additional (shorter) logical simulation times can be found in [9]. However, apart from slightly higher variation in the results, there were no significant differences between the different runtimes.

Evaluating the SimQPN Solver, for both processing resource utilizations and usage scenario throughputs, it showed predictions within 2% of the SimuCom results. Mean response time predictions showed a deviation of under 12%. The analysis overhead compared to SimuCom was reduced by over 90% in most cases, and over 50% for the CoCoME case study. The major reduction in analysis time is not surprising. SimuCom saves the context state for each request in the system individually while SimQPN (and LQNS/LQSim) encodes the state for groups of requests that share the same path through the system.

LQSim could only handle the ABB Demonstrator case study, where it showed results very close to LQNS for all metrics. As LQNS runs much faster than LQSim and could handle all but the SPECjEnterprise2010 case study, only LQNS is discussed here. For processing resource utilizations, LQNS stayed within 2% of the SimuCom results. For usage scenario throughputs, it stayed within 9%. For mean response times, however, errors of over 70% were observed. LQNS, being an analytical method, generally runs about an order of magnitude faster than the SimQPN Solver.

Considering the limitations of LQNS and LQSim in regard to the chosen case studies, a comparison is still worthwhile as the approaches are strongly related. It can also be assumed that the PCM-to-LQN transformation can be improved to allow the analysis of most of the provided scenarios.

TABLE II
RESULTS ACCURACY

Metric	SimQPN relDiff	LQNS relDiff	LQSIM relDiff
SPECjE (A)			
U_{WLS_CPU}	0.153%	-	-
U_{DBS_CPU}	$\approx 0\%$	-	-
$R_{Scenario}$	1.46%	-	-
SPECjE (B)			
U_{WLS_CPU}	-0.142%	-	-
U_{DBS_CPU}	$\approx 0\%$	-	-
$R_{Scenario}$	-0.294%	-	-
ABB			
U_{AS_CPU}	-0.224%	-0.0223%	$\approx 0\%$
U_{AS_CPU}	-0.113%	$\approx 0\%$	$\approx 0\%$
$R_{ScenarioA}$	9.16%	71.2%	77.8%
$R_{ScenarioB}$	1.06%	28.7%	29.3%
MediaStore			
U_{AS_CPU}	1.31%	1.34%	-
U_{DB_CPU}	-0.73%	-0.599%	-
$R_{Scenario}$	-1.12%	-10.3%	-
$X_{Scenario}$	1.14%	8.05%	-
CoCoME			
U_{AS_CPU}	$\approx 0\%$	0.176%	-
$R_{Scenario}$	$\approx 0\%$	-2.41%	-
$X_{Scenario}$	$\approx 0\%$	0.182%	-
BRS			
$U_{Server1\_CPU}$	$\approx 0\%$	-0.00759%	-
$U_{Server2\_CPU}$	$\approx 0\%$	-0.00634%	-
$U_{Server3\_CPU}$	$\approx 0\%$	$\approx 0\%$	-
$U_{Server4\_CPU}$	$\approx 0\%$	$\approx 0\%$	-
$R_{Scenario}$	11.2%	45.7%	-

TABLE III
RESULTS AVERAGE ANALYSIS TIMES (S)

Model	SimuCom	SimQPN	LQNS	LQSIM
SPECjE (A)	395.67	16.62	-	-
SPECjE (B)	616.50	30.83	-	-
ABB	125.01	9.77	0.82	5.17
MediaStore	8.88	2.02	0.36	-
CoCoME	4931	2387	5.73	-
BRS	183	16.3	9.32	-

In terms of the results stability, the solvers mostly showed comparable results. The coefficient of variation (CoV) remained under 2% for processing resource utilizations and throughputs. With the exeption of one ABB variant, in which the CoV for the mean response time reached 5.14%, it did not exceed 2%. For the analysis times, the CoV remained under 10%.

Overall, the SimQPN Solver provided very satisfactory results for all models and metrics. This means that results were obtained much faster while sacrificing only little accuracy at the same time. This shows, at least in the context of the numerous case studies, that the abstractions of the transformation are well-chosen and that QPNs and the available tools are well-suited for automated performance analysis of component-based software architecture models.

VII. Conclusions and Future Work

In this paper, we presented a mapping from PCM to QPNs, a solver tool implementing the mapping, and a detailed analysis of the accuracy and overhead of the solver compared to the existing solvers: SimuCom, LQNS and LQSim. The mapping was evaluated in the context of five representative case studies. The new 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. The LQN-based solvers showed significantly less accurate results regarding mean response times.

The use of PCM in more complex and new arising contexts will likely lead to scenarios that use many more of the PCM features, especially PCM stochastic expressions of higher complexity. To adequately support these upcoming scenarios, the limitations regarding the mapping of arbitrary stochastic expressions to common probability distributions need to be understood in more depth. Another topic requiring more research is the computation of response time distributions using token identities in QPNs [9].

With the presented approach, performance predictions for component-based software systems have become more flexible. More scenarios can be evaluated at a high level of accuracy in less time. Decisions involving the replacement of a component, or involving changes in the assembly, deployment and usage behavior of an adequately modeled system, can be made faster and can be supported by more information.

References

[1] 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.

[2] H. Koziolek and R. Reussner, "A model transformation from the palladio component model to layered queueing networks," in *Proc. on SIPEW '08*, 2008, pp. 58–78.

[3] H. Koziolek, "Parameter dependencies for reusable performance specifications of software components," Ph.D. dissertation, University of Karlsruhe (TH), 2008.

[4] 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.

[5] S. Kounev and A. Buchmann, "Performance modelling of distributed e-business applications using queuing petri nets," in *Proc. on ISPASS '03*, 2003, pp. 143–155.

[6] ——, "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.

[7] S. Kounev and C. Dutz, "QPME - A Performance Modeling Tool Based on Queueing Petri Nets," *ACM SIGMETRICS Performance Evaluation Review*, vol. 36, no. 4, pp. 46–51, 2009.

[8] F. Bause, P. Buchholz, and P. Kemper, "QPN-tool for the specification and analysis of hierarchically combined queueing petri nets," in *Proc. on MMB '95*, 1995, pp. 224–238.

[9] P. Meier, "Automated Transformation of Palladio Component Models to Queueing Petri Nets," Master's thesis, Karlsruhe Institute of Technology (KIT), 2010.

[10] H. Koziolek, "Performance evaluation of component-based software systems: A survey," *Performance Evaluation*, 2009.

[11] A. D. Marco and R. Mirandola, "Model transformation in software performance engineering," in *QoSA*, 2006.

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

[13] J. Henss, "Performance prediction for highly distributed systems," in *Proc. on WCOP '10*, vol. 2010-14. Karlsruhe Institue of Technology, 2010, pp. 39–46.

[14] *Unified Modeling Language (UML) Specification 2.1.2*, OMG Std., 2007.

[15] G. P. Gu and D. C. Petriu, "XSLT transformation from UML models to LQN performance models," in *Proc. on WOSP '02*, 2002, pp. 227–234.

[16] S. Bernardi, S. Donatelli, and J. Merseguer, "From UML sequence diagrams and statecharts to analysable petri net models," in *Proc. on WOSP '02*, 2002, pp. 35–45.

[17] H. Gomaa and D. Menascé, "Performance engineering of component-based distributed software systems," in *Performance Engineering*, ser. LNCS, 2001, vol. 2047, pp. 40–55.

[18] X. Wu and M. Woodside, "Performance modeling from software components," *SIGSOFT Softw. E. Notes*, vol. 29, no. 1, pp. 290–301, 2004.

[19] *UML Profile For Schedulability, Perf., And Time 1.1*, OMG Std., 2005.

[20] *UML Profile for Modeling and Analysis of Real-Time and Embedded Systems Beta 2*, OMG Std., 2008.

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

[22] M. Marzolla and S. Balsamo, "UML-PSI: The UML performance simulator," *Quantitative Eval. of Syst.*, vol. 0, pp. 340–341, 2004.

[23] M. Tribastone and S. Gilmore, "Automatic extraction of PEPA performance models from UML activity diagrams annotated with the MARTE profile," in *Proc. on WOSP '08*, 2008.

[24] A. Bertolino and R. Mirandola, "CB-SPE tool: Putting component-based performance engineering into practice," in *Component-Based Software Engineering*, ser. LNCS, 2004, vol. 3054, pp. 233–248.

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

[26] F. Bause, "Queueing Petri Nets – a formalism for the combined qualitative and quantitative analysis of systems," in *Proc. on Petri Nets and Performance Models*, 1993, pp. 14–23.

[27] G. Bolch, S. Greiner, H. De Meer, and K. S. Trivedi, *Queueing Networks and Markov Chains: Modeling and Performance Evaluation with Computer Science Applications*, 2nd ed. J. Wiley & Sons, 2006.

[28] *Meta Object Facility (MOF) 2.0 Query/View/Transformation Specification 1.0*, OMG Std., 2008.

[29] H. Perros, *Computer simulation techniques – the definitive introduction*. http://www.csc.ncsu.edu/faculty/perros/simulation.pdf: E-Book, NC State University, 2003.

[30] F. Brosig, S. Kounev, and K. Krogmann, "Automated extraction of palladio component models from running enterprise java applications," in *P. on VALUETOOLS '09*, 2009, pp. 1–10.

[31] Q-ImPrESS project results. http://www.q-impress.eu/wordpress/documentation/deliverables/.

[32] 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. LNCS, 2008, vol. 5153, pp. 16–53.

[33] A. Martens, H. Koziolek, S. Becker, and R. H. Reussner, "Automatically improve software models for performance, reliability and cost using genetic algorithms," in *Proc. on WOSP/SIPEW '10*, 2010, pp. 105–116.