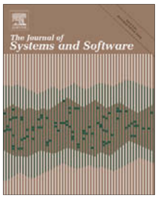
The Journal of Systems and Software 82 (2009) 3–22

Contents lists available at ScienceDirect

The Journal of Systems and Software

journal homepage: www.elsevier.com/locate/jss



The Palladio component model for model-driven performance prediction

Steffen Becker [a,*], Heiko Koziolek [b], Ralf Reussner [a]

[a] *IPD, University of Karlsruhe, FZI Forschungszentrum Informatik, Haid-und-Neu-Strasse 10-14, 76131 Karlsruhe, Germany*
[b] *Graduate School TrustSoft, University of Oldenburg, 26111 Oldenburg, Germany*

ARTICLE INFO

Article history:
Received 28 May 2007
Received in revised form 22 February 2008
Accepted 7 March 2008
Available online 23 May 2008

Keywords:
Component-based software engineering
Software architecture
Performance prediction

ABSTRACT

One aim of component-based software engineering (CBSE) is to enable the prediction of extra-functional properties, such as performance and reliability, utilising a well-defined composition theory. Nowadays, such theories and their accompanying prediction methods are still in a maturation stage. Several factors influencing extra-functional properties need additional research to be understood. A special problem in CBSE stems from its specific development process: Software components should be specified and implemented independently from their later context to enable reuse. Thus, extra-functional properties of components need to be specified in a parametric way to take different influencing factors like the hardware platform or the usage profile into account. Our approach uses the Palladio component model (PCM) to specify component-based software architectures in a parametric way. This model offers direct support of the CBSE development process by dividing the model creation among the developer roles. This paper presents our model and a simulation tool based on it, which is capable of making performance predictions. Within a case study, we show that the resulting prediction accuracy is sufficient to support the evaluation of architectural design decisions.

1. Introduction

In CBSE, a central idea is to build complex software systems by assembling basic components. The initial goal of CBSE was to increase the level of reuse. However, composite structures may also increase the predictability of the system during early design stages, because certified models of individual components can be composed, enabling software architects to reason on the composed structure. This is important for functional properties, but also for extra-functional properties like performance (i.e., response time, throughput, resource utilisation) and reliability (i.e., mean time to failure, probability of failure on demand).

Prediction methods for performance and reliability of general software systems are still limited and rarely used in industry (Balsamo et al., 2004; Becker et al., 2006). Especially for component-based systems further challenges arise. Opposed to object-oriented system development and performance prediction (Smith and Williams, 2002), where developers design and implement the whole system, several independent developer roles are involved in the creation of a component-based software system. Component developers produce components that are assembled by software architects and deployed by system allocators. The diverse information needed for the prediction of extra-functional properties is thus spread among these developer roles.

Most existing methods for component-based performance prediction require software architects to model the system based on specifications of single components. It is often assumed that the software architect can provide missing information. This assumption is necessary because of today's incomplete component specifications. For example, in Bertolino and Mirandola (2004) software architects model the control flow through the component-based architecture, which is impossible if components are black boxes and the dependencies between provided and required interfaces are unknown. Thus, a special component specification is needed.

Other approaches neglect factors affecting the perceived performance of a software component like influences by external services (Eskenazi et al., 2004; Sitaraman et al., 2001), changing resource environments (Bondarev et al., 2005; Hamlet et al., 2004; Liu et al., 2005), or different input parameters (Bertolino and Mirandola, 2004). However, for accurate predictions, all these dependencies have to be made explicit in component specifications. This is only partially possible in approaches based on the UML and the SPT profile (OMG, 2005), which does not provide facilities to specify parameter dependencies, does not explicitly support different developer roles, and complicates the implementation of model transformations as discussed in this paper.

[*] Corresponding author. Tel.: +49 721 9654 612; fax: +49 721 9654 623.
E-mail address: sbecker@fzi.de (S. Becker).

0164-1212/$ - see front matter © 2008 Elsevier Inc. All rights reserved.
doi:10.1016/j.jss.2008.03.066

With the Palladio[1] component model (PCM), a meta-model allowing the specification of performance-relevant information of a component-based architecture, we provide an original and innovative approach to address the identified problems. First, our model is designed with the explicit capability of dividing the model artifacts among the different roles involved in a CBSE development process. These artifacts can be considered as instances of domain specific modelling languages (DSL), which capture the information available to a specific developer role. Secondly, the model reflects that a component can be used in changing contexts with respect to the components it is connected to, the allocation of the component on resources, or different usage contexts. This is done by specifying parametric dependencies, which allow deferring context decisions like assembly or allocation. By this, our design and analysis meta-models are the first ones, which *explicitly include all* factors influencing the performance of a software component, namely the factors: implementation, performance of external services, performance of execution environment and usage profile.

For an initial validation, we have developed a tool capable of simulating instances of the PCM to obtain performance metrics. We used this tool in a case study to simulate the performance of a component-based online shop. Comparing the simulation results with measurements made on an implementation of the architecture enabled estimating the accuracy of our simulations.

The contributions of this paper are

(i) a component meta-model for quality-of-service (QoS) predictions and
(ii) an according performance simulation including a case study.
(iii) A discussions on the weaknesses of UML for model-driven performance prediction and
(iv) a discussion of using UML versus a dedicated meta-model for component-based performance prediction.
(v) A discussion of applying different prediction techniques at different stages of software development.

The PCM (a) is based on our CBSE role concept (Koziolek and Happe, 2006), (b) allows parametric QoS specifications which include a comprehensive set of influencing factors on component performance and (c) supports arbitrary stochastic distribution functions to specify component behaviour as well as to predict QoS properties. Instances of the meta-model are simulated with a simulation tool specialised for the features of the PCM based on model transformations. This paper extends the work presented in Becker et al. (2007) by (a) a new simulation approach for PCM instances based on model transformations, (b) a thorough comparison of UML versus PCM and (c) the discussion of various alternative prediction techniques for the PCM with their strengths and weaknesses.

This paper is structured as follows: In Section 2, we review related work. Section 3 introduces our CBSE role concept and provides details of the component meta-model. Examples of how the parametric dependencies can be specified and evaluated are given in Section 4. We provide extensive tool support for modelling PCM instances as described in Section 5. Section 6 details on the developed simulation tool before Section 7 compares the simulation approach with other evaluation methods. A discussion of different shortcomings of the UML for component-based performance prediction follows in Section 8. Assumptions and limitations of our work are discussed in Section 9. In Section 10, a case study applying our simulation tool to a model instance is presented. Finally, we conclude our paper and outline future work.

2. Related work

The Palladio component model is related to CBSE, performance prediction methods, usage modelling, and simulation approaches.

Although the concept of reusable software components has been discussed since the famous 1968 NATO conference on software engineering (McIlroy, 1968), CBSE has gained wide-spread attention only since the 1990s due to the failure of object-oriented programming to effectively support reuse (Cheesman and Daniels, 2000; Heineman and Councill, 2001; Szyperski et al., 2002). The PCM follows Szyperski's definition of contractually specified, independently deployable software components (Szyperski et al., 2002) and has extended the component-based development process model of Cheesman and Daniels (2000) with model-driven performance predictions (Koziolek and Happe, 2006).

A number of *component models* have been developed to support component-based development (Lau and Wang, 2005). They are related to architectural description languages (ADL) (Medvidovic and Taylor, 2000). Industrial component models, such as EJB, COM, or CCM, support specific implementation tasks and do not deal explicitly with extra-functional properties. They lack component concepts established in academia, such as explicit composite components or protocol-enhanced interfaces. Academic component models serve specific analysis purposes (e.g., Koala, SOFA, Kobra, ROBOCOP, PECT, COMQUAD, PECOS, Pin) like interoperability checking or analysing effects of runtime reconfiguration. From these, PECT (Hissam et al., 2002), ROBOCOP (Bondarev et al., 2004), and COMQUAD (Goebel et al., 2004) explicitly deal with extra-functional properties.

However, their support for context-independent QoS specification of components is limited. PECT's component specifications do not allow adaptation for different resource environments, as the resource demands are specified as timing values measured on a specific platform. The same is true for ROBOCOP, which however allows the specification of timing values dependent to constant input parameters. The PCM additionally allows dependencies to output parameters to model more complex component interaction and stochastic characterisations of parameter values, which provide more refined results than constants. COMQUAD allows the component developer to provide any kind of performance annotation, therefore the resulting specification by different developers can easily be incompatible, which reduces their reusability.

For *performance predictions*, researchers have created many analytical models, such as queueing networks (Denning and Buzen, 1978; Jain, 1991; Kleinrock, 1975; Lazowska et al., 1984; Menasce et al., 2004), Markov chains (Trivedi, 1982), layered queueing networks (Rolia and Sevcik, 1995; Woodside et al., 1995), stochastic Petri nets (Bause and Kritzinger, 1996), and stochastic process algebras (Hermanns et al., 2002). These models mainly focussed on accurate modelling of system resources before Smith et al. highlighted the influence of software with their software performance engineering (SPE) approach (Smith, 1990, 2002).

The *SPE community* has put much effort into creating designer-friendly performance modelling notations, which can be transformed into the analytical models mentioned before (comprehensive survey in Balsamo et al. (2004)). These efforts led to defining performance related extensions to UML as the de facto standard modelling language, namely the UML SPT profile (OMG, 2005), and the UML MARTE profile (OMG, 2006), which is still under development. Recently, several meta-models of the performance domain have emerged from the SPE community (Cortellessa, 2005), namely SPE–MM (Smith et al., 2005), CSM (Petriu and Woodside, 2004), and KLAPER (Grassi et al., 2007) providing a solid basis for model transformations. The PCM meta-model

[1] Our component model is named after the Italian renaissance architect Andrea Palladio (1508–1580), who, in a certain way, tried to predict the aesthetic impact of his buildings in advance.

includes concepts similar to these meta-models, but focusses on component-based software architectures.

Several approaches for *component-based performance prediction* have been proposed during the last decade (survey in Becker et al. (2006)). Measurement-based approaches (Chen et al., 2005; Denaro et al., 2004; Kounev, 2006; Liu et al., 2005; Yacoub, 2002) use existing systems or prototypes to measure performance properties and calibrate performance models with the results. Performance analysts may then use the models to analyse the effects of changed workloads or faster hardware with low effort. Model-based approaches, such as the PCM (Bertolino and Mirandola, 2004; Bondarev et al., 2005; Bondarev et al., 2004; Eskenazi et al., 2004; Gomaa and Menasce, 2001; Hamlet et al., 2004; Hissam et al., 2002; Sitaraman et al., 2001; Wu and Woodside, 2004) also support creating performance models from scratch.

Some of these approaches target the performance specification of reusable components like the PCM, but often neglect single influencing factors (such as external service calls, usage profile, and specifics of the execution environment). Sitaraman et al. (2001) use extended Big-O notations to specify performance of software components depending on input parameters, but neglect parameters of calls to required services. Their specification language furthermore does not support external service calls. Gomaa and Menasce (2001) require component developers to specify fixed timing values for resource usages and probabilities for control flow branches. However, both properties may depend on parameter values at runtime, which are unknown to component developers at design time. Instead, the PCM allows component developers to specify parameter dependencies for these properties.

Bertolino and Mirandola (2004) provide a component performance specification parametrised over the execution environment. This specification does not make calls by components to required services explicit and does not include any notion of service parameter dependencies. Hamlet et al. (2004) modelled software components as functions transforming values. They measured how these components propagate inputs to other components to gain accurate performance predictions. In the PCM, component developers can specify these input propagations so that software architects do not have to measure components, which may not be available for testing.

The component performance specifications based on layered queueing networks modelled by Wu and Woodside (2004) allow resource demands parameterised for different execution environments. These specifications may include dependencies to parameters, but these are not specialised stochastic characterisations as in the PCM. The parameterisation of branch probabilities or loop iteration numbers is not foreseen in that approach as well as the parameterisation of other parameter characterisations, which is possible in the PCM. Eskenazi et al. (2004) do allow parameter dependencies to control flow branches in a simple manner, but assume components without external service calls.

Finally, many performance analysts use *simulation* techniques to evaluate their models. Simulation approaches from the SPE community can be found in Arief and Speirs (2000), Balsamo and Marzolla (2003), Cortellessa et al. (2007), de Miguel et al. (2000). However, they do not target component-based software architectures. We used the Desmo-J framework (DESMO, 2008) to implement our performance simulation. Other tools supporting simulation are CSIM (2007), Opnet (Chang, 1999), SPE-ED (Smith and Williams, 2002), umlPSI (Balsamo and Marzolla, 2003), and SSJ (L'Ecuyer and Buist, 2005).

3. Component-based performance modelling

The PCM is a meta-model for the description of component-based software architectures. The model is designed with a special focus on the prediction of QoS attributes, especially performance and reliability. In this paper, we focus on the performance related parts of the PCM. In the following, we give some details on our envisioned CBSE development process and the participating roles. Afterwards, we highlight some concepts of our meta-model omitting concepts not used in this paper.

3.1. CBSE development process

In the CBSE development process (see also (Koziolek and Happe, 2006)), we distinguish four types of developer roles involved in producing artefacts of a software system (see Fig. 1).

- *Component developers* specify and implement the components. The specification contains an abstract, parametric description of the component and its behaviour.
- *Software architects* assemble components to build applications. For the prediction of extra-functional properties, they retrieve component specifications by component developers from a repository.

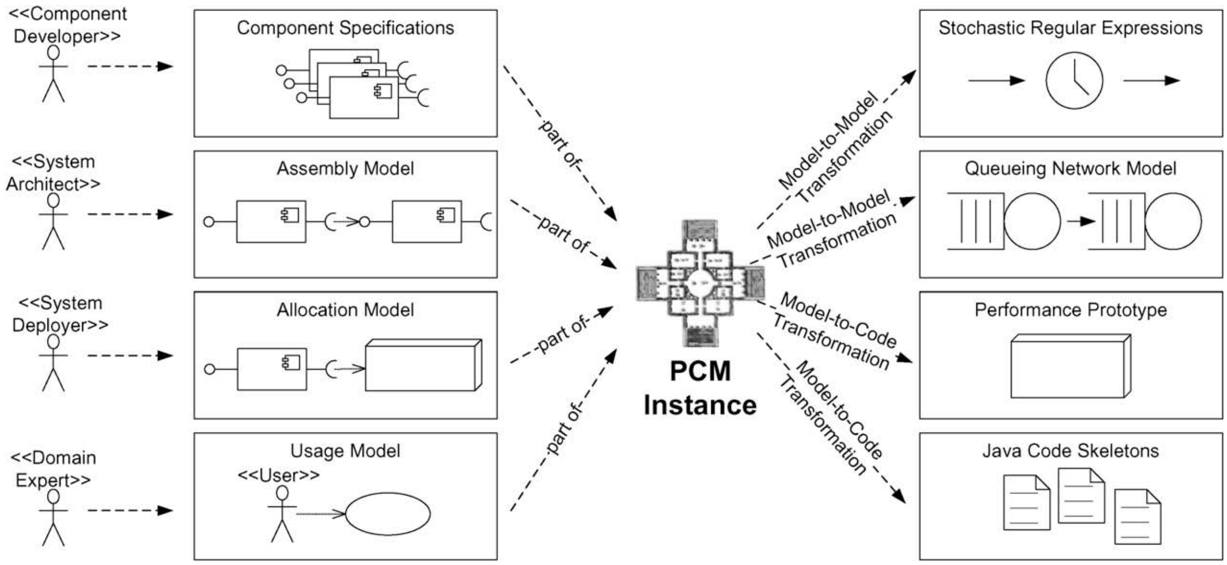


Fig. 1. Process.

- *System deployers* model the resource environment and afterwards the allocation of components from the assembly model to different resources of the resource environment.
- Business *domain experts*, who are familiar with the customers or users of the system, additionally provide usage models describing critical usage scenarios as well as typical parameter values.

The complete system model can be derived from the partial models specified by each developer role and then extra-functional properties can be predicted. Each developer role has a domain specific modelling language and only sees and alters the parts of the model in its responsibility. The introduced partial models are aligned with the reuse of the software artefacts.

3.2. Component specification (component developers)

Component developers specify and implement components. They deposite development artefacts, such as models and code, into repositories (Section 3.2.1). Additionally, they may compose basic (atomic) components to composite components. For performance predictions, they provide service effect specifications (SEFFs) to abstractly describe service behaviour (Section 3.2.2).

3.2.1. Repository

Fig. 2 shows an example of a PCM repository with modelling artefacts. In general, component developers specify components via provided and required interfaces. Component A requires interface Interface1, which is provided by component B. An interface serves as contract between a client requiring a service and a server providing the service. Components implement services specified in their provided interfaces and may use services specified in their required interfaces during execution.

Interfaces are first-class entities in the PCM and thus exist independently from components. Interfaces include a list of service signatures. Interface1 includes signatures for service1 and service2. Interfaces are themselves neither providing nor requiring. Only their relations to components define their roles. We call this relation provided role or required role.

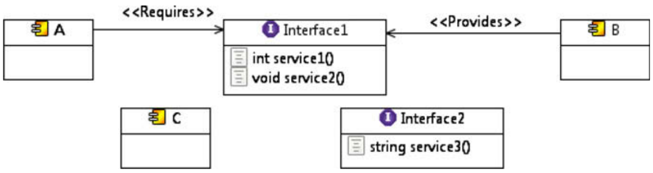


Fig. 2. Repository example.

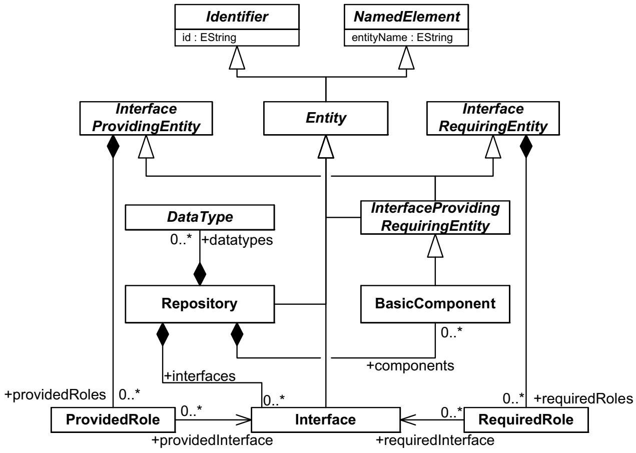


Fig. 3. Repository meta-model.

Fig. 3 depicts the meta-model for repositories, which is part of the component developer's domain specific language. Repositories contain BasicComponents, Interfaces, and DataTypes. Entities, such as Interfaces and Repositories, have an id and a name. BasicComponents are Entities requiring and providing interfaces modelled by the ProvidedRoles and RequiredRoles.

Interfaces (cf. Fig. 4) reference their direct parentInterfaces and indirect ancestorInterfaces. They contain a set of Signatures. Signatures have a name, reference a data type as return type, and include a list of parameters, which again reference DataType. The PCM supports PrimitiveDataTypes (e.g., INT, BOOL, CHAR), CollectionDataTypes (e.g., arrays, sets, trees of inner data types), and CompositeDataTypes, which contain inner (primitive or collection) data types. These data types can be mapped to Java or C-Sharp data types.

3.2.2. Service effect specification

Service effect specifications describe the relationship between provided and required services of a component. Besides performance predictions, service effect specifications were used to automatically adapt component protocols (Reussner, 2003). In this application, service effect specifications were finite state machines which described an abstraction of the control flow where transitions denoted external service calls. This model was extended with transition probabilities to predict the probability of failure on demand of component services (Reussner et al., 2003). The ResourceDemandingServiceEffectSpecification (RDSEFF) used for performance prediction as discussed in this article built on this idea of a probabilistic abstraction of the control flow, but differ from the service effect specifications of Reussner et al. (2003). Firstly, RDSEFFS use a notation stemming from UML activity diagrams, i.e., activities are denoted by nodes. Secondly, the resource demand per activity can be specified and thirdly, dependencies of transition probabilities and resource demands on the formal parameters of the service can be specified. This model is discussed in detail below.

To each provided service of a component, developers can add a so-called ResourceDemandingServiceEffectSpecification (RDSEFF). It describes

- how the service uses hardware/software resources;
- how the service calls the component's required services.

Resource demands in RDSEFFs abstractly specify the consumption of resources by the service's algorithms, e.g., in terms of CPU

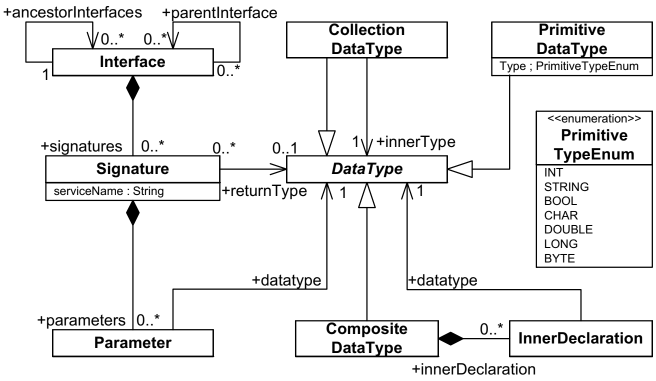


Fig. 4. Interface meta-model.

units needed or bytes read or written to a hard disk. Resource demands as well as calls to required services are included in an abstract control flow specification, which captures call probabilities, sequences, branches, loops and forks.

RDSEFFs abstractly model the externally visible behaviour of a service with resource demands and calls to required services. They present a grey box view of the component, which is necessary for performance predictions, because black box specifications (e.g., interfaces with signatures) do not contain sufficient information. RDSEFFs are not white box component specifications, because they abstract from the service's concrete algorithms and do not expose the component developer's intellectual property. Component developers specify RDSEFFs during or after component development and thus enable performance predictions by third parties. Software architects do not need to understand RDSEFFs, as performance analysis tools encapsulate them.

To get an initial idea of RDSEFFs, consider the example in Fig. 5. The left hand side depicts the simplified code of the service execute. The right hand side shows the corresponding RDSEFF. It includes calls to required services as ExternalCallActions, and abstracts computations within the component's inner method into an InternalAction. Control flow constructs are modelled only between calls to required services, while control within the internal computations is abstracted. The example includes parametric dependencies on the branch transitions and the number of loop iterations, which we will explain in Section 4.

An single InternalAction can potentially subsume thousands of instructions into a single modelling entity as long as these instructions do not interact with other components and perform only component-internal computations. In many cases, an RDSEFFs consists only of a few InternalActions and ExternalActions while at the same time modelling large amounts of code.

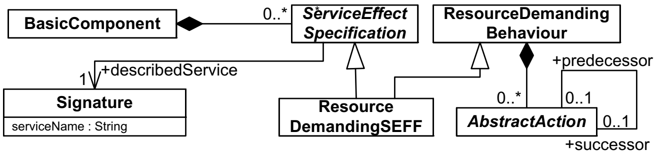


Fig. 6. RDSEFF (1/5): relation between components and behaviour.

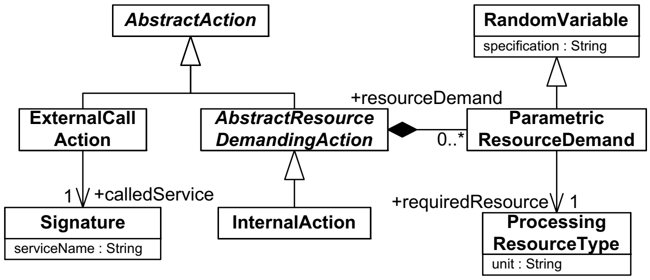


Fig. 7. RDSEFF (2/5): ExternalCallActions and InternalActions.

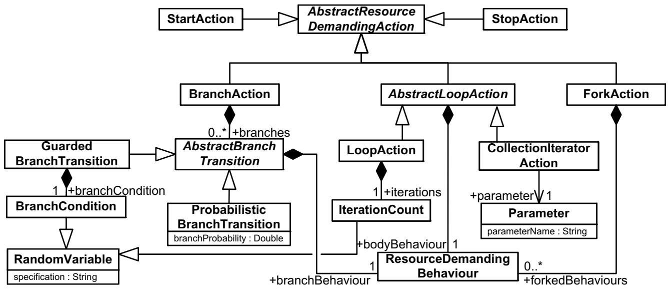
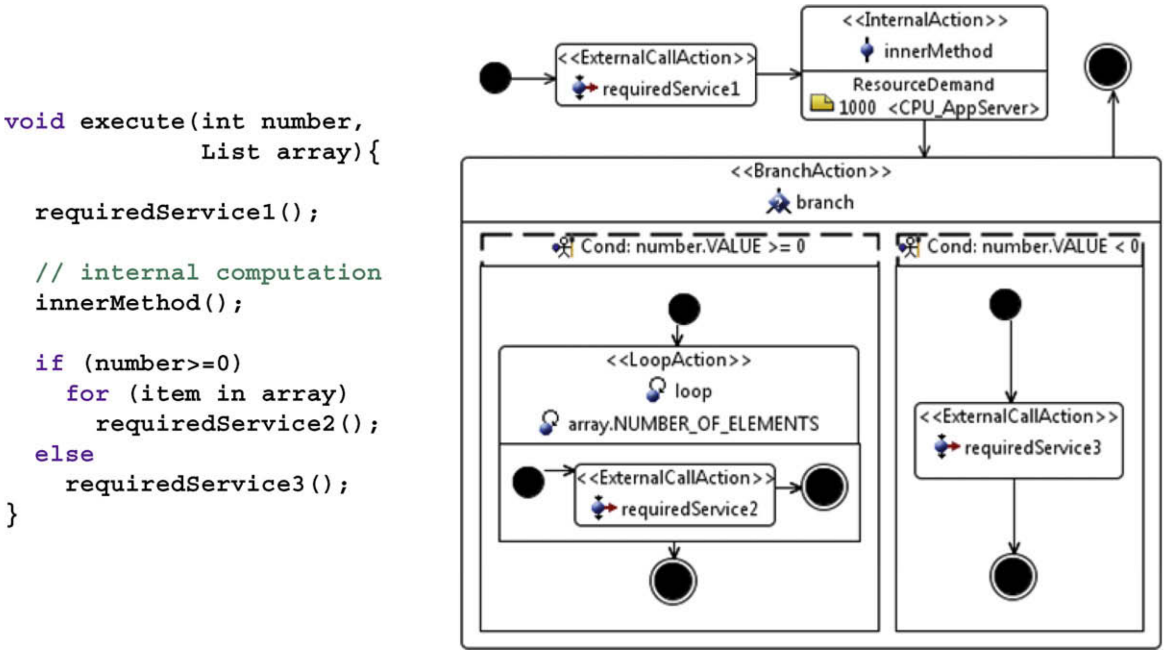


Fig. 8. RDSEFF (3/5): control flow primitives.

Figs. 6–10 depict the RDSEFF meta-model, which is part of the component developer's DSL. We spread the description over multiple figures for clarity. Fig. 6 shows the relation between components and RDSEFFs. A BasicComponent contains a number of ServiceEffectSpecifications, which each reference the signature of the service they describe. ResourceDemandingSEFF inherits both from ServiceEffectSpecification and ResourceDemandingBehaviour. The latter includes a number of AbstractActions, which model the service's behaviour as a chain of steps. Each action may reference a predecessor and successor.



```
void execute(int number,
             List array){

  requiredService1();

  // internal computation
  innerMethod();

  if (number>=0)
    for (item in array)
      requiredService2();
  else
    requiredService3();
}
```

Fig. 5. Source code and RDSEFF example.

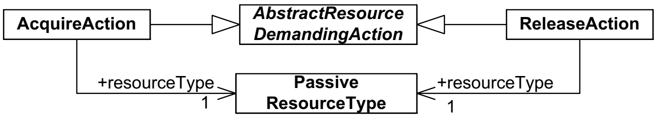


Fig. 9. RDSEFF (4/5): Using passive resource types.

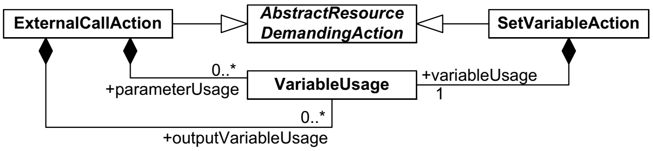


Fig. 10. RDSEFF (5/5): `VariableUsage`.

`AbstractActions` are either `ExternalCallActions` or `AbstractResourceDemandingActions` (Fig. 7). The former model calls to the component's required service and therefore references the called service's signature. The latter can place loads on the resources the component is using (e.g., CPU, memory, storage device, network connection, etc.). Therefore, `AbstractResourceDemandingActions` contain so-called `ParametricResourceDemands` to specify the amount of resources needed (e.g., 256 CPU units). These demands inherit from `RandomVariable` enabling the specification of constants (e.g., 1024 Bytes), probability distributions (e.g., in 40% 1000 CPU units and in 60% 2000 CPU units), or functions of random variables (e.g., 2000 CPU units + 100 CPU units per byte of input parameter). Examples for parametric resource demands follow in Section 4.

As a concrete resource demanding action, an `InternalAction` potentially combines a large number of operations in a single model entity. The algorithms executed internally within the `InternalAction` are not visible in the RDSEFF to create an abstraction of the code and avoid a white-box specification. Notice, that `ExternalCallActions` cannot contain resource demands, as the resource demands of the required services have to be specified in the RDSEFFs of that required services.

Each `ParametricResourceDemand` references a `ProcessingResourceType`. These are abstract resource *types* (e.g., CPU, HD, Network) instead of concrete resource *instances* (e.g., 5 Ghz CPU, 20 MB/s HD, 1000 MBit/s Network), because component developers can and should not know during component specification the actual resources the component will later be deployed on. The PCM decouples component specification and concrete resource specification, so that independent developer roles can execute these tasks. Once the system deployer specifies the resource environment (cf. Section 3.4), which contains processing times for different resource types, it can be combined with RDSEFFs to convert the abstract resource demands into concrete timing values.

RDSEFFs support several constructs to model control flow primitives (Fig. 8). `StartActions` start the action chain of a `ResourceDemandingBehaviour` and have no predecessor, while `StopActions` end the action chain and have no successor. Additionally, RDSEFFs may contain branches, loops, and forks.

`BranchActions` model XOR control flow alternatives and contain a number of `AbstractBranchTransitions`, which can be either `GuardedBranchTransitions` or `ProbabilisticBranchTransitions`. The component developer may use the former to specify a `BranchCondition` depending on an input parameter, whose actual value is unknown during component specification (see example in Section 4). Once the actual value is known from the usage model, a transition probability can be associated with the branch. `ProbabilisticBranchTransitions` directly include a branch probability without a dependency to an input parameter. Although most components do not include such a probabilistic control flow, this model construct is a convenient way for the component developers to specify the transition probability in case the dependency to input parameters is unknown or hard to derive from the code. Both kinds of branch transitions contain a `ResourceDemandingBehaviour` (see also Fig. 6), which includes actions executed in the body of the branch.

RDSEFFs allow two kinds of loops: `LoopAction` and `CollectionIteratorAction`. Both contain an inner `ResourceDemandingBehaviour` to model actions of the loop body. The former models repetitive behaviour and includes the number of loop iterations as a `RandomVariable`. It is thus easily possible to specify a constant or arbitrary distribution function for the number of loop iterations. Other models (such as Markov chain based approaches) allow specifying a probability p of entering a loop and a probability $1 - p$ of exiting a loop. However, this binds the number of loop iterations to a geometrical distribution, which seldomly occurs in reality. Therefore, the RDSEFF meta-model only permits explicit modelling of loops and does not allow backward references in the chain of actions within a `ResourceDemandingBehaviour`, which could be used to introduce loops with a geometrically distributed number of iterations.

`CollectionIteratorActions` are a special construct for loops, in which the number of repetitions depends on the size of a collection. Thus, these actions reference an input parameter of the service, which must be a collection. The number of loop iterations equals the number of elements within the collection. This model construct is similar to expansion nodes in UML 2 activities.

`ForkActions` model AND control flow alternatives, meaning that the inner forked `ResourceDemandingBehaviours` execute concurrently. The successor action of the `ForkAction` is not executed until all forked behaviours have terminated.

Besides using processing resources such as CPU and memory, RDSEFFs may also use passive resources such as threads, semaphores, or database connections from a pool. These resources are usually available in a limited number (e.g., in a thread or buffer pool). A service has to acquire a passive resource to continue execution and release it again after using it. `AbstractResourceDemandingActions` can therefore be specialised into `AcquireActions` and `ReleaseActions`, which reference `PassiveResourceTypes` (Fig. 9). Passive resource usage may have a significant influence on the execution time of a service due to waiting times, and has therefore been included in the PCM.

The service modelled by the RDSEFF may use input parameters when calling required services. If these input parameters can influence the resource demands of the required services, the component developer should model them explicitly. Therefore `ExternalCallActions` may contain a number of `VariableUsages` to model characterisations of input parameters (Fig. 10). We will explain `VariableUsages` and the corresponding parameter characterisations in Section 3.5. Furthermore, the return values of a required service may influence its resource demands. Hence, `ExternalCallActions` include a second set of `VariableUsage` to store output parameter characterisations of required services.

To characterise the output of the service, component developers may use `SetVariableActions`. As we allow output parameters besides service return values, using multiple `SetVariableActions` is possible. It might occur that the same output variable is set to different values within different branches of the control flow. In this case, the output variable gets assigned the last value set by `SetVariableAction`.

3.3. Architecture model (software architect)

Software architects connect components and their roles to build `ComposedStructures`, i.e., component-based software architectures. Component developers may also build ComposedStructures, to create composite components. However, only software architects can embed a ComposedStructure into a System, which defines the modelled system's boundaries, can be allocated to a resource environment (Section 3.4), and exposes services to end-users (Section 3.5).

Fig. 11 shows an example System including a ComposedStructure. As software architects can use multiple component instances of the same component type in the same system (notice component A), components are embedded in unique AssemblyContexts (visualised by the dashed lines), which are referenced by AssemblyConnectors. An AssemblyConnectors connects a RequiredRole of a component with a ProvidedRole of another component. This means that any call emitted by the component requiring a service of the RequiredRole is directed to the connected component providing that service. For AssemblyConnectors it is important that the required and provided interfaces match (respecting subtyping), e.g., that the service is provided as expected by the requiring component. Matching of interfaces is performed based on the Interface instances in a PCM Repository where the interfaces and its inheritance are specified. These Interfaces are referenced directly by the connectors. DelegationConnectors bind roles provided (required) by the ComposedStructure with provided (required) roles of components within the ComposedStructure.

We illustrate the meta-model of ComposedStructures in Fig. 12. Software architects use this DSL to create component architectures, while component developers use it to create composite components, which they store in Repositories like BasicComponents. ComposedStructures contain a set of AssemblyContexts, which embed a single (basic or composite) component. The software architect can bind AssemblyContexts using AssemblyConnectors, which reference two AssemblyContexts (providing and requiring), and the respective ProvidedRole and RequiredRole. A ComposedStructure publishes its own provided and required roles and binds them to inner component's provided and required roles via ProvidedDelegationConnectors and RequiredDelegationConnectors.

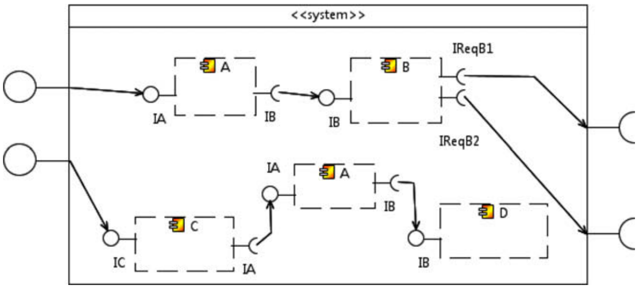


Fig. 11. System example.

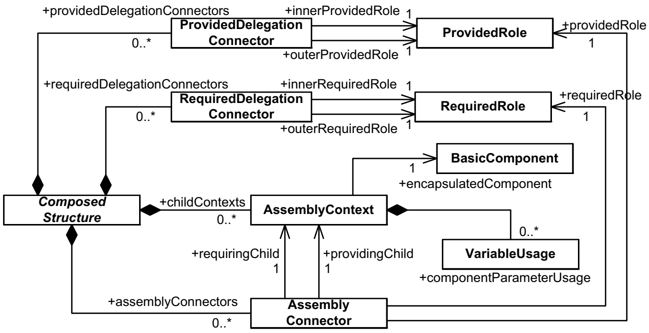


Fig. 12. Composed structure meta-model.

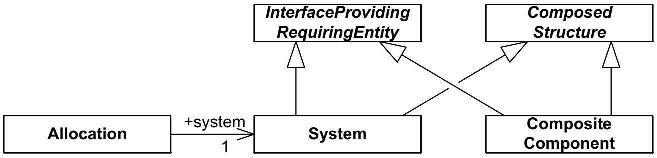


Fig. 13. System meta-model.

Fig. 13 shows the difference between System and ComposedStructure in the meta-model. A System is (like a basic component) an InterfaceProvidingRequiringEntity and thus may contain provided and required roles. It furthermore inherits from ComposedStructure (unlike basic component) as its internal structure consists of a set of components. An Allocation from the system deployer references a system and maps the contained inner components to concrete resources. CompositeComponent inherits from the same classes as System, but does not contain a reference from Allocation as these components can only be deployed after embedding them into a System.

3.4. Resource model (system deployer)

System deployers model the resource environment of the component-based software architecture and allocate individual components to resources. In the PCM, they instantiate abstract resource types from a global resource repository to describe their concrete resources. Component developers in turn provide RDSEFFs referencing only resource types without knowing concrete resource instances. System deployers group resources in resource containers, for example to model a server as a resource container with a CPU, memory, and cache as inner resources. They link resource containers with communication resources, which can model network connections.

The example in Fig. 14 shows a simple resource environment consisting of two resource containers. They contain multiple concrete resources (e.g., CPU, thread pool, etc.) with performance specifications (e.g., processing speed, throughput, capacity, etc.) and are connected with a linking resource modelling a FastEthernet connection.

Fig. 15 shows the meta-model for the abstract resource type repository, concrete resource environment, and the allocation of

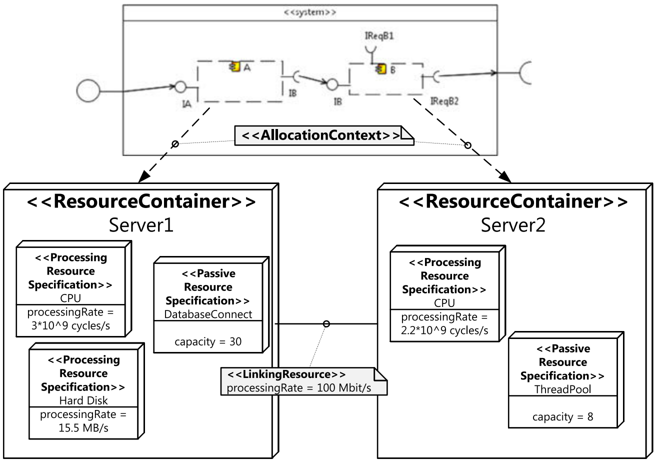

<<system>>
IReqB1
A
B
IA
IB
IB
IReqB2
<<AllocationContext>>
<<ResourceContainer>>
Server1
<<Processing Resource Specification>>
CPU
processingRate = 3*10^9 cycles/s
<<Passive Resource Specification>>
DatabaseConnect
capacity = 30
<<Processing Resource Specification>>
Hard Disk
processingRate = 15.5 MB/s
<<LinkingResource>>
processingRate = 100 Mbit/s
<<ResourceContainer>>
Server2
<<Processing Resource Specification>>
CPU
processingRate = 2.2*10^9 cycles/s
<<Passive Resource Specification>>
ThreadPool
capacity = 8

Fig. 14. Resource environment and allocation example.

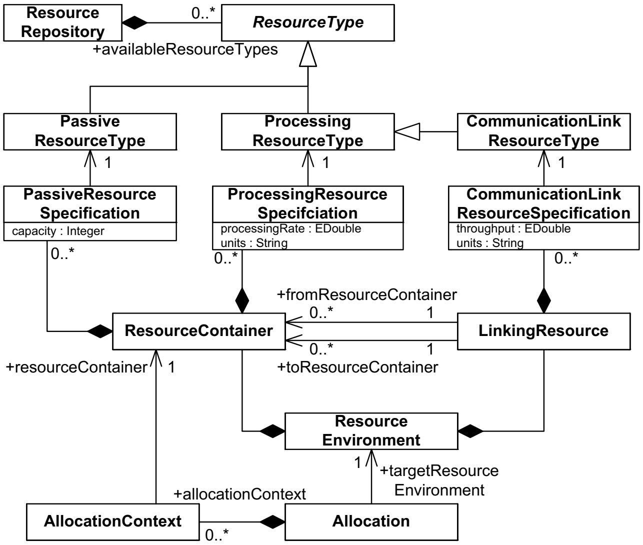


Fig. 15. Resource meta-model.

components to resources. It is the system deployer's domain specific modelling language.

A ResourceRepository contains a number of ResourceTypes. The resource repository has to be specified globally (i.e., for a modelled systems), so that component developers and system deployers can refer to the same types of resources. So far, the PCM coarsely distinguishes between ProcessingResourceTypes (e.g., CPU, HD, Memory, etc.), PassiveResourceTypes (e.g., semaphores, etc.), and CommunicationLinkResources, which are specialised ProcessingResourceTypes to model network connections. In the future, we will extend this simple model with special modelling constructs for middleware (such as caches) and operating system resources (such as schedulers). The general resource model (GRM) from the UML SPT profile (OMG, 2005) is a more refined resource model, which might inspire extension to the PCM resource model.

System deployers specify concrete ResourceEnvironments, which contain a number of ResourceContainers connected by LinkingResources. ResourceContainers may include ProcessingResourceSpecifications or PassiveResourceSpecifications. Both model concrete resources and reference the abstract corresponding resource types. They include specific characteristics about the resource like the processing rate (e.g., instructions per time unit, bytes read per time unit) or the capacity (e.g., the size of a passive resource pool).

Once the system deployer has specified the resource environment based on the available resource types, components can be allocated to the concrete resources using an Allocation. It references a System and a ResourceEnvironment and contains a number of AllocationContexts, which associate an AssemblyContext with a ResourceContainer. The abstract resources referenced by the RDSEFFs included in the assembly context's components can then be substituted by the concrete resources from the resource environment to compute actual resource demands.

3.5. Usage model (domain expert)

Domain experts specify a system's usage in terms of workload (i.e., the number of concurrent users), user behaviour (i.e., the control flow of user system calls), and parameters (i.e., abstract characterisations of the parameter instances users utilise). Software architects may also construct usage models from requirements documents.

Fig. 16 includes an example usage model. It specifies a closed workload of 15 concurrent users, who repeatedly executed the modelled steps and begin again at the start node after 1 second of think time after reaching the end node. The behaviour is expressed as a control flow graph similar to UML2 activities, but additionally contains transition probabilities on branches and the number of iterations on loops. Via variable characterisations, the domain expert can abstractly characterise the parameter values of the users as random variables. Note, that unlike the RDSEFF, the usage model does not contain any parametric dependencies or resource demands, as these are concepts referring to component behaviour but not to user behaviour.

Figs. 17–19 depict the PCM usage meta-model. An UsageModel contains a number of UsageScenarios, which each model single use cases of the system. An UsageScenario contains a Workload describing the usage intensity and a ScenarioBehaviour, which models the control of user actions. Workload may be either open or closed, analog to workloads in queueing networks (Lazowska et al., 1984). OpenWorkloads do not fix the number of users, but

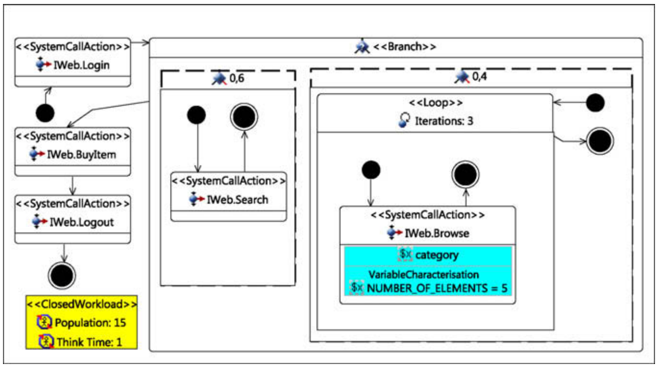


Fig. 16. Usage model example.

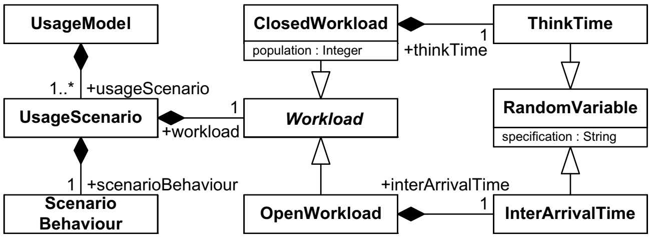


Fig. 17. Usage model (1/3): Usage Scenario and Workload.

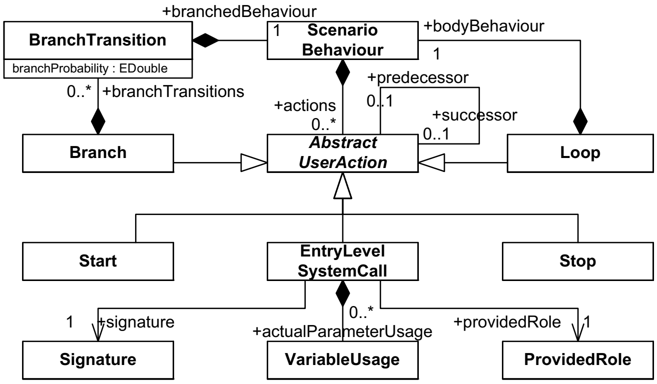


Fig. 18. Usage model (2/3): Scenario behaviour.

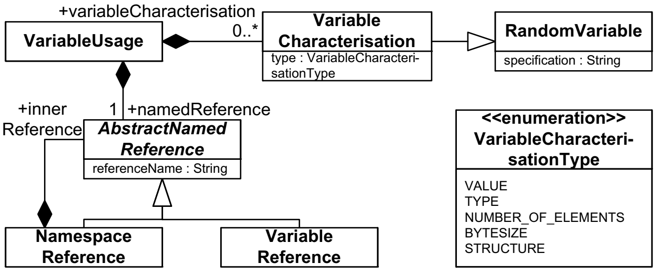


Fig. 19. Usage model (3/3): Variable usage.

model an InterArrivalRate of users as a RandomVariable. For example, a system could receive 5 user requests per second. ClosedWorkloads model a fixed number of users (i.e., population), who unlimitly circulate within the system. They execute an UsageScenario from start to stop, and then reenter the scenario at the start node after the specified ThinkTime.

A ScenarioBehaviour (Fig. 18) contains a number of AbstractUserActions, which model the user behaviour. Besides Start and Stop nodes, they may contain Branches with branch probabilities and Loops with the number of iterations as RandomVariable (not depicted here). Domain experts can model call of services at system provided roles via EntryLevelSystemCalls. Therefore, they reference a Signature and a ProvidedRole. Note that users cannot invoke services of components within a System that are not delegated to system provided roles. EntryLevelSystemCalls can contain a number of VariableUsages to abstracly characterise the parameter values of the users.

VariableUsages (Fig. 19) contain the name of a variable (AbstractNamedReference) and a number of VariableCharacterisations for this name. Names consists of NamespaceReferences and VariableReferences. Namespaces are necessary for characterising the inner elements of collection or composite parameters. VariableCharacterisations are modelled as RandomVariables for a fixed set of characterisation types (VALUE, TYPE, NUMBER_OF_ELEMENTS, BYTESIZE, STRUCTURE).

For example, domain experts can characterise the *value* of an integer variable with a probability distribution to specify the different parameter values of larger user groups stochastically. They may also specify the *bytesize* of a parameter as a constant or distributed random variable if it influences performance. The variable characterisation types have to be chosen according to their influence on performance properties.

4. Parametric dependencies

The performance of a software component is influenced by its *usage* (Hamlet et al., 2004). The resource demand may vary depending on input parameters (e.g., uploading larger files with a component service produces a higher demand on hard disk and network). Different required services can be called as a result of different inputs, thus the branch probabilities in the SEFF are most often linked to the usage profile (e.g., required service A is called if some integer parameter is larger than zero, otherwise service B is called). Furthermore, the parameters passed to required services (forming a usage model for the required component) may also depend on a service's own input parameters.

The central dilemma of the component developer is that during component specification it is unknown how the component will be used by third parties. Thus, in case of varying resource demands or branch probabilities depending on user inputs, the component developer cannot specify fixed values. However, to help the software architect in QoS predictions, the component developer can specify the *dependencies* between input parameters and resource demands, branch probabilities, or loop iteration numbers in RDSEFFs. If a usage model of the component has been specified by domain experts or if the usage of the component by other components is known, the actual resource demands and branch probabilities can be determined by the software architect by solving the dependencies.

In the PCM, we use *random variables* to express resource demands or numbers of loop iterations. Mathematically, a random variable is defined as a measurable function from a probability space to some measurable space. More detailed, a random variable is a function $X : \Omega \to \mathbb{R}$ with Ω being the set of observable events and $\mathbb{R}$ being the set associated to the measurable space. Observable events in the context of software models can be for example response times of a service call, the execution of a branch, the number of loop iterations, or abstractions of the parameters, like their actual size or type.

A random variable X is usually characterised by stochastical means. Besides statistical characterisations, like mean or standard deviation, a more detailed description is the probability distribution. In the discrete case, a probability distribution yields the probability of X taking a certain value, which is often abbreviated by $P(X = t)$. It can be specified by a probability mass function (PMF), as used in our component model. The event spaces Ω we support include integer values $\mathbb{N}$, real values $\mathbb{R}$, boolean values and enumeration types (like "sorted" and "unsorted").

Additionally, it is often necessary to build new random variables using other random variables and mathematical expressions. For example, to denote that the response time is 5 times slower, we would like to simply multiply a random variable for a response time by 5 and assign the result to a new random variable. For this reason, our specification language supports some basic arithmetic operations ($*,-,+,/,\ldots$) as well as logical operations for boolean type expressions ($==,>,<,$AND,OR,...).

We use the introduced random variables in the following to provide several examples for specifying dependencies between input parameters and QoS-related specifications. We also use random variables in the case study to characterise the parameters of the calls issued to the system.

4.1. Branch conditions

In Fig. 20, the ResourceDemandingSEFF of the service HandleShipping from an online-store component is depicted. It has been specified by a component developer in a parametrised form. The

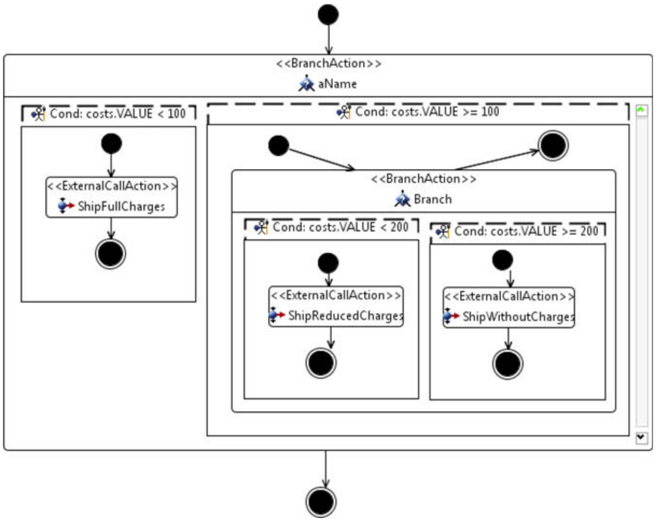


Fig. 20. Branch condition example.

service calls required services shipping a customer's order with different charges depending on its costs, which it gets passed as an input parameter. If the order's total amount is below 100 Euros, the service calls a service preparing a shipment with full charges (ShipFullCharges). If the costs are between 100 and 200 Euros, the online-store grants a discount, so ShipReducedCharges is called. Orders priced more than 200 Euros are shipped for free with the ShipWithoutCharges service.

The `ResourceDemandingSEFF` in Fig. 20 is an abstract representation of the control flow through the component. Internal computations as well as resource demands of the service have been abstracted, because they are not relevant for QoS analysis in this case.

Once a domain expert specifies the value of the parameter `costs`, it can be derived which of the services will be called. In this example, the domain expert has specified the value as a PMF (Table 1) according to a customer analysis. The PMF is used in order to represent a whole group of customers using the service. It has been specified within a `UsageModel`, which is not illustrated here for brevity. In this case, the sample space Ω of the PMF consists of integer values $\mathbb{N}$ representing the costs of an order. Note, that the specification of the domain expert only refers to parameters visible at the service interface. The usage specifications can be made without referring to internals of the component thus preserving the black box principle.

To determine the branch probabilities which are required for the QoS analysis, random variables associated with the branches have to be evaluated. For the first branch node, let A and B denote the left and right branch conditions, where A is the event that costs are below 100 Euros (costs.VALUE <= 100) and B the event that costs are larger than 100 Euros (costs.VALUE $>$ 100). With the usage profile provided by the domain expert, their probabilities of becoming true can be computed as: $P(A) = 0.35 + 0.25 = 0.6$ and $P(B) = 0.20 + 0.15 + 0.05 = 0.4$.

After any branching event X has occured, the sample space Ω on subsequent nodes is restricted to $\Omega' = \Omega \cap X$. This has to be considered by the following evaluations. In the example, this situation occurs at the second branch node. Let C and D denote the left and right branch conditions, where C is the event that the costs are below 200 Euros and D the event that the costs are larger than 200 Euros. When computing the probabilities, it has to be taken into account, that the sample space has been restricted by B. Thus, a new sample space $\Omega/ = \Omega \cap B$ has been created. Using the usage profile from the domain expert, the conditional probabilities are computed as: $P(C|\Omega') = P(C \cap \Omega')/P(\Omega') = 0.35/0.4 = 0.875$ and $P(D|\Omega') = P(D \cap \Omega')/P(\Omega') = 0.05/0.4 = 0.125$.

4.2. Loop iterations

In the PCM, it is possible to assign a number of iterations to a loop. This can be done in a parameterisable form, as illustrated by the following example. Fig. 21 shows the `ResourceDemandingSEFF` of the service UploadFiles. It gets an array of files as input parameter and calls the external service HandleUpload within a loop for each file. As an example, the component shall be used in a new architecture for an online music repository. Users shall upload music albums via the service, which are then stored one by one in a database that is connected to the service HandleUpload.

Table 1
Probability of costs; specified by domain expert

Costs (Euro)	Probability
0–49	0.35
50–99	0.25
100–149	0.20
150–199	0.15
200-	0.05

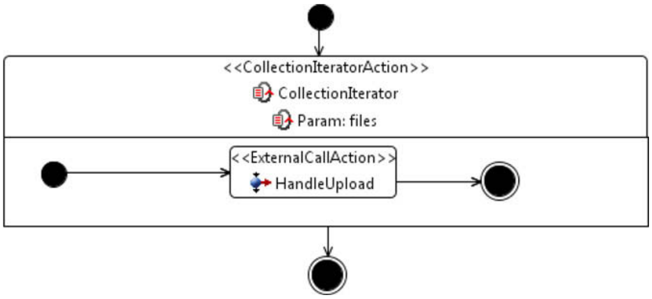


Fig. 21. Loop example.

Table 2
Probability for number of files; specified by domain expert

Number of files	Probability
8	0.1
9	0.1
10	0.2
11	0.4
12	0.2

A domain expert has analysed user behaviour and found that users usually upload albums with 8–12 music files. Thus, a PMF for the number of files in the input parameter files has been specified (Table 2).

With the specified dependency to the number of elements in the input collection, the probability distribution of random variable X_{iter} for the number of loop iterations in the `ResourceDemandingBehaviour` can be determined. The required service HandleUpload will be called 8 times (probability 0.1), 9 times (0.1), 10 times (0.2), and so on (see Table 2). If the dependency had not been specified, it would not have been known from the interfaces how often the required service would have been called. Thus, with the specified PMF, a more refined prediction can be made for varying usage contexts.

4.3. Parametric resource demand

Besides branch conditions and loop iterations, resource demands can be specified in a parameterised form. In many cases, this is the most influencing factor for varying response times. In Fig. 22, the component service ProcessFile(byte[] file) receives an input parameter file, which is processed internally. The component developer has specified that the resource demand of this service depends on the size of the input file (file.BYTESIZE assigned by the method's caller), in particular 3 CPU operations are executed for each byte of the file.

Because of varying file sizes due to varying usages, the domain expert has specified a PMF for the size of the input file. After analysing a large number of usage traces from a similar system, a fine grained distribution function could be specified, which is shown in Fig. 23. Note, that in this example the size of the file is the only

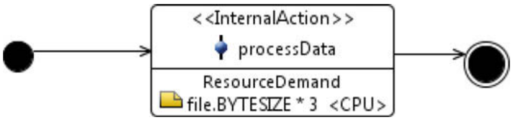


Fig. 22. Resource demand example.

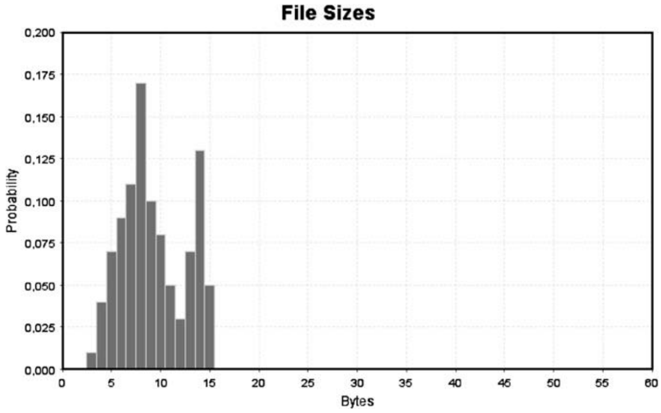


Fig. 23. File sizes (PMF), from domain expert.

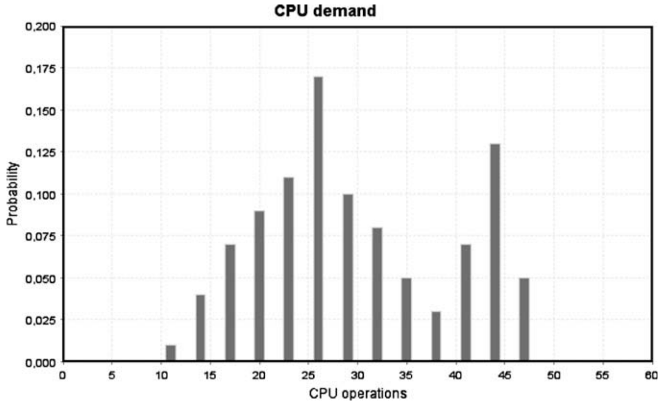


Fig. 24. CPU operations (PMF), computed.

attribute relevant for the QoS analysis. Other attributes of the file parameter, such as the value or the type of the file, are irrelevant in this case and do not need to be specified. This is an example of abstracting unnecessary details from the model and in many cases such abstractions model reality well enough to make sufficiently accurate predictions.

To compute the actual resource demand on the CPU from the specification in Fig. 22, the underlying PMF can be obtained by multiplying the PMF for the file sizes by a factor of 3, thus stretching the PMF as depicted in Fig. 24. Instead of file sizes, the PMF now denotes the probabilities for the number of CPU operations, which are executed for the given usage context. Once the actual CPU is known from the allocation to the resource environment, to which the component is deployed, the time for executing service ProcessFile can be computed after specifying the execution time for a single CPU operation.

4.4. Parametric parameter usage

When calling required services, component services may pass parameters. In many cases, these parameters can be fixed in the implementation. However, sometimes, input parameters for required service calls actually depend on the provided service's own input parameters. In the PCM, such a dependency can be expressed by attaching a `VariableUsage` to an `ExternalCallAction`.

As an example, in Fig. 25, the `ResourceDemandingSEFF` of the service SendCompressed is shown. It receives a `file` as an input parameter, compresses it using a ZIP algorithm, and then passes it to another component by calling the service SendFile. The component developer has specified that the compression reduces the size of the input files by 50%. If a domain expert specifies the input file size, or if it has been specified in the `VariableUsage` of another component calling this component, the file size for the input parameter of the `ExternalCallAction` can be determined. If the file size has been specified as a PMF (like in the previous example), its domain values have to be multiplied by a factor of 0.5, thus contracting the PMF.

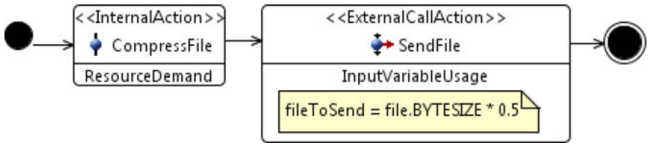


Fig. 25. Propagating input parameters example.

5. Tool support

Tool support is crucial for the specification and analysis of PCM instances. We provide an integrated tool set based on the Eclipse platform. Our tool combines modelling, performance prediction based on the models, skeleton code-generation for later implementation of the modelled system and reverse engineering of code into model instances into a single IDE. It is expected that the integration of the modelling process into the development environment leads to a broader applicability of performance prediction during early development cycles.

5.1. Modelling support

Our IDE offers graphical editors to create model instances. The editors have been generated using the graphical modelling framework (GMF) framework (see Fig. 26) and supplemented with manual code additions. These graphical editors visualise a UML-like concrete syntax for PCM instances to increase developer acceptance. To avoid the ambiguities of UML, our syntax is more restrictive than UML. For example, control flow links cannot link backwards in a RDSEFF which forces component developers to make branches and loops explicit. In addition to graphical modelling, our tools also conveniently support entering performance annotations. Special textual dialogs utilise the grammar and semantics of our stochastic expressions to offer context-sensitive code completion and on-the-fly error reporting.

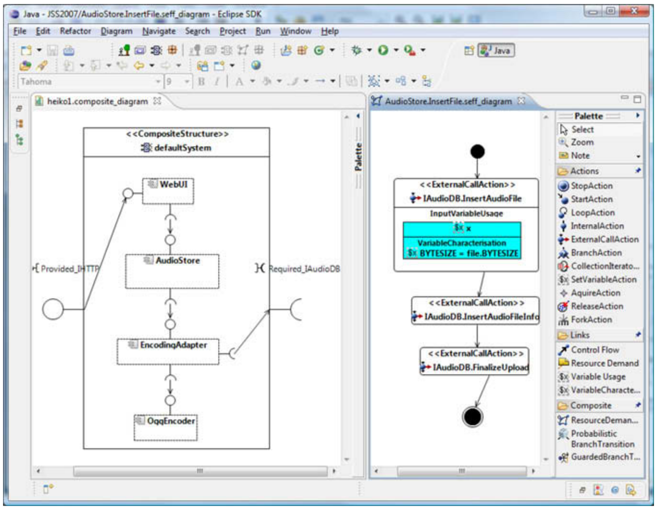
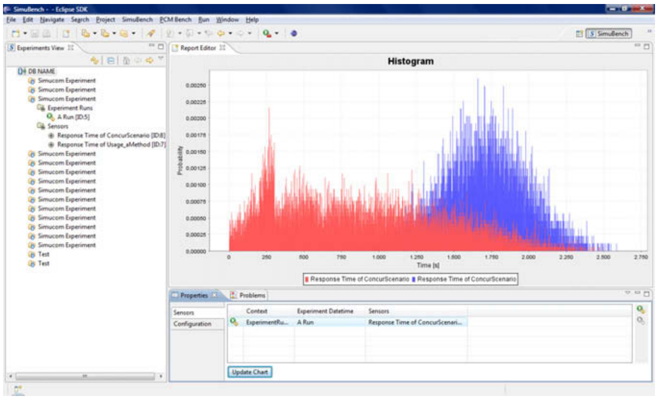


Fig. 26. PCM bench: Modelling perspective.



Context	Experiment Datetime	Sensors
ExperimentRu...	A Run	Response Time of ConcurScenari...

Fig. 27. PCM bench: Analysis perspective.

5.2. Performance prediction

The developer can start performance analyses directly from the model editors due to their integration into Eclipse. The tools fully encapsulate both model-2-model and model-2-code transformations, which relieves the developer from the burden of transforming and analysing models manually. Section 6 gives details of the internals of the model-drivens simulation used in the case study of this paper. Additionally, there is an analytical solver based on stochastic regular expressions capable of solving single-user scenarios involving arbitrary distribution functions. Details on this analytical solver are out of the scope of this paper and can be found in Koziolek et al. (2007). Section 7 compares capabilities of the simulation and the analytical solver.

5.3. Results visualisation

A framework with charting capabilities plots histograms or cumulative distribution functions of predicted response times and supports the software architect in drawing conclusions on the expected performance of a modelled architecture (See Fig. 27). Statistical data, such as mean values, standard deviations, etc. are provided with an interface to the statistics package R. We also use this interface to compare predictions and measured values by means of statistical tests to validate our prediction models.

5.4. Model-2-code transformations

Our toolset contains two model-2-code transformations. The first generates a J2EE implementation prototype which is ready to be deployed. When executed, the prototype records real performance metrics by issuing dummy demands according to its PCM specification on its environment replacing estimates of the processing speed with real measurements. Current support for resources is limited to dummy CPU demands, however, other resource types will be added soon.

The second transformation generates code skeletons for component developers as well as for the software architect. If some components in the modelled architecture have only been specified but not been implemented, this transformation provides a convenient way to start their implementation. The current model-2-code transformation generates source code for the J2EE platform including deployment descriptors and build scripts. The component developer needs to add missing business logic to the code skeletons generated from RDSEFFs. Afterwards, the components can be deployed on a J2EE application server.

Section 7 also discusses conducting performance measurements with code generated by the two model-2-code transformations. It relates measurements to predictions with our analytical approach and the simulation presented in the next section.

5.5. State of the tool

Our tools have undergone a maturation phase of six month during the preparation phase of an empirical case study with students (Martens, 2007). The stable version is available from the PCM's homepage (Palladio, 2007). Development of the tool continues in the publicly available unstable branch.

6. SimuCom simulation framework

To evaluate the model concepts introduced in the previous sections, we have built a simulation tool. This tool takes an instance of the PCM as an XMI serialisation and builds a simulation in order to get the response times under the specified workloads. We validate whether the meta-model is appropriate and can be used to model an example system, e.g., if all necessary model concepts are available. Furthermore, by comparing the predicted values to measured values of an implemented architecture, we can evaluate if the introduced parametric specifications can be used to give realistic predictions. The latter are presented in the next section. This section briefly gives some details on the implementation of our simulation.

6.1. Technical foundation

In contrast to the initial presentation of our simulation framework in Becker et al. (2007), we changed the analysis of a PCM instance's performance from a visitor-based, interpreter approach to a generative approach for shorter simulation runtime. Our current version generates Java simulation code from PCM instances using a model-2-code transformation with the openArchitectureWare framework (oAW, 2007). The simulation's core is based on the discrete-event Java simulation framework Desmo-J (DESMO, 2008). The implementation utilises features by the Eclipse environment. The model-2-code transformation generates Java source code of an OSGi plugin, which is compiled using Eclipse's Java Development Tools (JDT). A controller GUI dynamically loads the compiled plugin into the OSGi runtime and starts executing the simulation. After finishing a simulation run, the controller GUI unloads the plugin again and cleans up the generated files.

6.2. Simulation run

The workloads specified in the model instance are transformed into simulated workload drivers, which simulate the specified workload scenario. For this, the workload drivers execute a generated thread per arriving user. The threads implementation corresponds to the usage behaviour as given in the model instance. Depending on the type of workload, i.e., closed workload or open workload, the workload's semantics is different. For closed workloads, the behaviour of each simulated user is executed, then the workload driver waits for the given think time and afterwards starts from the beginning. For open workloads, the workload driver starts a user thread in the frequency of the specified arrival rate.

For each component in the model, code is generated implementing the component. For `BasicComponents` a class is generated containing the components services and their implementation as given by their corresponding `RDSEFF`. In this code, `RDSEFF`'s control flow constructs map directly to their corresponding Java constructs and the `RDSEFF`'s `ExternalCallAction` map on direct method invocation. For `InternalActions`, the generated code loads the respective resource with the resource

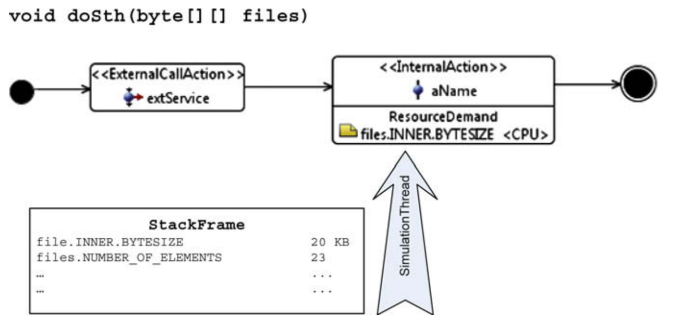


Fig. 28. A simulation thread and the simulated stack frame.

demand as determined by the stochastic expression describing the demand.

Several specifications in the SEFF like number of loop iterations, branch conditions, and resource demand depend on other random variables, i.e., parameter abstractions. A sample of this random variable is generated by the evaluation of the definition as outlined in Section 4. For this, the specification of the random variable is evaluated. For any random variable in the parse tree, the simulation framework's random number generators are used to get samples. Mathematical operations are evaluated using their normal semantics. The result of the evaluation is the desired sample of the depending random variable.

For each ProcessingResource, the simulation instantiates a framework class implementing a request queue and an accompanying scheduler. The scheduling strategy is configurable, the current implementation supports the FIFO and processor sharing strategies. Threads requesting the processing of demands get delayed with respect to simulation time until the demand is processed.

In order to get simulation results, generated probes exist in the simulation code which record execution times and queue lengths during the simulation run. The results of the run are stored in memory for immediate analysis or in a relational database for later evaluation.

6.3. Simulated stack frame

Special treatment is required for parameter abstractions, since they are only valid during the execution of a called method. To simulate this behaviour, we introduced simulated stack frames in alignment with the real execution of a software system.

Whenever an external call is to be simulated, a new simulated stack frame is built analogously to the method's stack frame. To initialize a stack frame, the random variables characterising the parameters of the called service are evaluated and the result is stored in the stack frame. It is then passed to the called behaviour (cf. Fig. 28). This way we simulate part of the usage context of the components relevant to performance.

When the simulated control flow returns to the caller, the evaluation is continued after the external call using the old call stack again.

6.4. Simulation end

The simulation comes to an end as soon as the PMF of the predicted workload scenario's execution time has been approximated in a way that further simulations of the workload scenario don't change the output PMF significantly anymore. One option of doing this is to evaluate the confidence interval of the mean's estimate of the resulting PMF. If the width of this interval drops below a certain configurable threshold after new simulated measurements are added, the simulation stops. However, as meeting the stop criteria might take a long time, additionally, an upper bound for the simulation run can also be given. Meanwhile the simulation supports live updates of the response time's distribution function, so that the software architect can watch the simulation proceed and cancel it in advance if the results are sufficiently precise. For example, when judging design alternatives often rough estimates of the resulting distribution function are sufficient for making a decision.

7. Comparison of different evaluation approaches

In Section 5, we briefly described four of the implemented model transformations for the PCM:

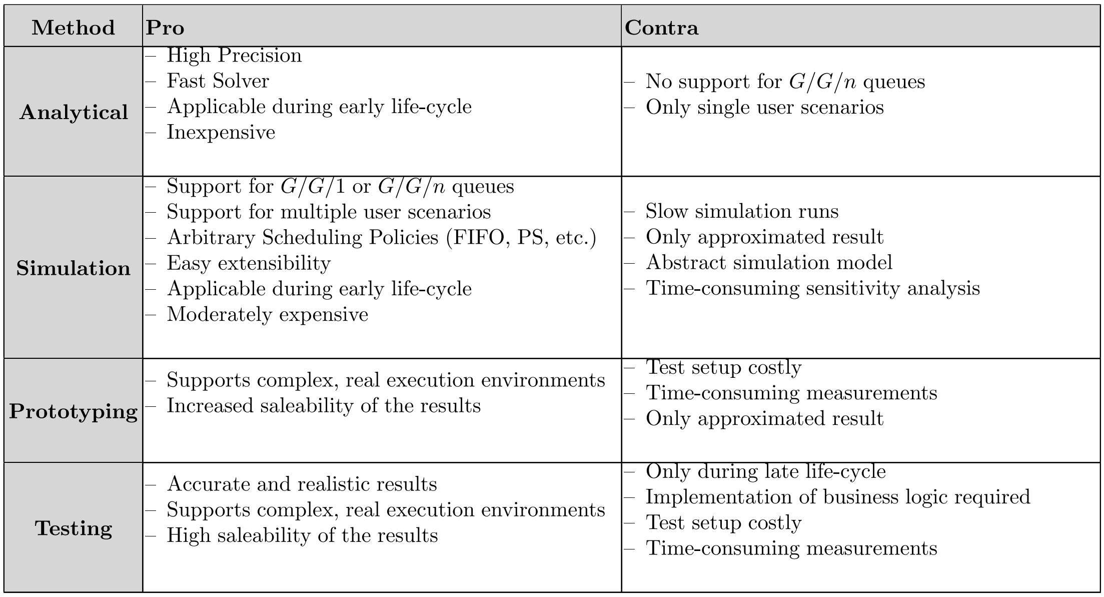
- a model-2-model transformation to an analytical solver based on stochastic regular expressions,
- a model-2-code transformation to a simulation model based on queuing networks,
- a model-2-code transformation to a performance prototype capable of producing resource demands on real resources, and
- a model-2-code transformation generating Java code skeletons to start an implementation.

In this section, we compare them with respect to their performance evaluation capabilities and argue why we used the simulation in this paper. An overview of the advantages and disadvantages of the transformations when used for performance evaluation is depicted in Fig. 29. A similar comparison can be found in Jain (1991).

The *analytical solver* for instances of the PCM only supports analysing single-user scenarios, because the PCM allows arbitrary distributions for resource demands and the inter-arrival rate of open workloads. For a queuing network, this implies generally distributed service times and generally distributed arrival rates at the service centers representing resources (i.e., $G/G/1$ or $G/G/n$ queues (Bolch et al., 1998)). For this kind of queuing network, no general analytical solution is known. However, if no resource contention is present in the network (i.e., only one user in the system), an efficient analytical solution for the distributions of the response time can be calculated. Such calculations can be very fast due to efficient convolution of arbitrary probability density functions with Fast Fourier Transformations (FFT). For the example in the case study of this paper they needed ca. 200 ms. Details of the method can be found in Koziolek et al. (2007).

The *simulation* based approach presented in this paper is able to deal with $G/G/n$ queues. In addition to standard queuing networks, our simulation supports network routing based on the specification in the RDSEFFs instead of probabilistic routes. The result of a simulation run contains response time distributions of each executed service. The simulation resolves resource contentions for the service centres either by a FIFO or processor sharing scheduling policy. Further scheduling policies including more realistic schedulers of today's operation systems and multi-core handling will be implemented in the future. The simulation can therefore predict the performance for more complex scenarios than the analytical solver. However, executing the simulation is more time-consuming (e.g., it took 8 minutes for this paper's simple case study). Although the simulation time was still sufficiently short for practical use in our examples, the time for executing the simulation with much larger examples remains unknown.

By executing the *performance prototype* generated via model-2-code transformation, the software architect can gain more accurate figures. The simulation's assumption that a resource demand can be evaluated by multiplying an abstract resource demand



Method	Pro	Contra
Analytical	– High Precision – Fast Solver – Applicable during early life-cycle – Inexpensive	– No support for $G/G/n$ queues – Only single user scenarios
Simulation	– Support for $G/G/1$ or $G/G/n$ queues – Support for multiple user scenarios – Arbitrary Scheduling Policies (FIFO, PS, etc.) – Easy extensibility – Applicable during early life-cycle – Moderately expensive	– Slow simulation runs – Only approximated result – Abstract simulation model – Time-consuming sensitivity analysis
Prototyping	– Supports complex, real execution environments – Increased saleability of the results	– Test setup costly – Time-consuming measurements – Only approximated result
Testing	– Accurate and realistic results – Supports complex, real execution environments – High saleability of the results	– Only during late life-cycle – Implementation of business logic required – Test setup costly – Time-consuming measurements

Fig. 29. Overview of prediction methods with their pros and cons.

(i.e., 1000 CPU instructions) with a processing rate (i.e., 1000 CPU instructions/ms) often does not hold any more in today's environments (Notice, that this computation does not include scheduling and multi-core effects which is handled by the queueing centres in the simulation). Several types of hardware would need much more fine-grained specifications for a realistic characterisation. For example, the speed of a CPU is influenced by its caching structure, pipelining algorithm, branch prediction, etc. Additionally, also the software environment may change. For example, server virtualisation results in varying processing speed per virtual server. Expressing all these factors in a performance model would result in a complex model, which might be unsolvable. The performance prototype can take these aspects into account. It requires that the dummy resource usage is not overly artificial compared to the final system. To execute the performance prototype, a complex experiment setup is needed, as the execution environment has to be created in a realistic manner. For the setup, an application server and client machines simulating the workload are needed. Measurements are performed in real time compared to simulation time in the simulation approach. Collecting a sufficient amount of measurements may take hours. For example, our case studies measurements took half a day.

Finally, the real system implemented using the code skeletons offers the real performance - no modelling is necessary here. However, in addition to the effort already needed for the prototype to setup and measure the performance figures, the time to implement the system has to be added. This approach of gaining performance values is only applicable in late life-cycles of the software system. When performance problems are discovered after the system has been implemented, a redesign and reimplementation is costly.

From the discussion, a trade-off becomes clear. The more realistic and accurate the predictions should be, the more time consumption and costs is involved. Based on analytical and simulation models, the software architect develops an initial system design. Only if this design offers sufficient performance figures, a prototype is generated to validate the figures in the real environment. Finally, the actual system is implemented.

8. UML2 versus PCM

In this section we provide rationale why we chose to define a new modelling language instead of extending UML2. The given discussion is compareable to the one published by Medvidovic et al. (2002). However, our discussion is more recent as it covers UML2 and its extended profiling mechanism and also recent advances in model-driven software development.

There are at least three alternatives for introducing new modelling constructs (Völter and Stahl, 2006):

(i) *Defining or extending a UML profile*: The UML provides a *lightweight* extension mechanism via profiles, which enable adapting UML models with stereotypes and tagged values.
(ii) *Extending the UML meta-model*: The UML meta-model can serve as the basis for a new modelling language (*heavy-weight* UML extension), which inherits classes from the UML meta-model and defines its own extensions by instantiating MOF classes.
(iii) *Defining a new MOF-based meta-model*: Finally, it is possible to define a new MOF instance, which serves as meta-model and is not tied to the UML meta-model. We chose this alternative when designing the PCM.

We will discuss the advantages and disadvantages of these three approaches in the following in order to justify our decision for a new MOF-based meta-model.

8.1. Defining or extending a UML profile

This is the most prominent way of introducing new modelling constructs and has many *advantages*. It extends a standardised modelling language with wide-spread practical use. Designers are familiar with the UML notation and can quickly incorporate the extensions. Many tools are available and established in the software industry, which enable editing UML models and also support using profiles. Designers have already modelled many software systems with UML, and they could extend the existing models.

However, this approach also bears several *disadvantages*. Existing UML diagrams are often unsuited for model transformations, because they usually have been defined for documentation and human communication and lack the formality needed for tool processing (Henderson-Sellers, 2005). Many UML diagrams only exist as Power Point slides or Visio diagrams with further textual, informal explanations and annotations. These diagrams require a significant effort to extract formal UML models and be prepared for machine interpretation and automated predictions in addition to the effort of extending them with a profile.

Defining a new UML profile is not necessary, because the existing UML profile for schedulability, performance, and time (SPT) (OMG, 2005) could be extended. This profile lacks constructs such as message sizes, time intervals, and characterisation of input parameters. The upcoming UML MARTE profile (OMG, 2006) may solve some of these deficits, but will not be finished until 2009. Furthermore, the SPT profile aims at object-oriented systems and lacks special means to specify the factors influencing the performance of software components.

While designers can use existing UML tools to add the SPT profile's stereotypes and tagged values to their models, support for entering the complex and error-prone tagged values for performance annotations is missing. Tagged values are strings in the SPT profile, so that syntax highlighting, code completion, and on-the-fly error reporting would be valuable for practical use. However, this would require proprietary extensions to existing UML tools.

Using a standardised model-2-model transformation language such as QVT (OMG, 2007) for processing UML models extended with the SPT profile is difficult, because QVT needs to process the abstract syntax of parsed tagged values. Therefore, processing these values requires additional, proprietary ad-hoc transformations (e.g., in Java), which decreases the advantage of using a standardised transformation engine.

Besides the deficits of the existing profiles and the described issues with UML's profiling mechanism, using profiles also implies inheriting UML's complexity and ambiguities, whose problems will be elaborated in the next paragraph.

8.2. Extending the UML meta-model

This approach has only seldomly been attempted in industry and academia. Changing and extending the UML meta-model implies many *disadvantages* such as losing standard conformity, tool support, and designer familiarity (some of them also apply for an own MOF-based meta-model, c.f. Section 8.3). Furthermore, this approach leads to inheriting the *complexity* and *ambiguity* of UML.

The *complexity* of UML2 (more than 1200 pages specification) is a major problem for model-driven approaches, because it complicates creating model transformations. For performance predictions, many UML constructs are not needed, because they have no counterparts in the performance domain (e.g., use cases diagrams, steps consuming no time, etc.). Model transformations have to ignore such model constructs. Performance analysis tools might not support many UML constructs, which would have to be excluded from the altered UML meta-model via constraints to create a sound approach. But due to the complexity of UML, defining OCL constraints for all unsupported constructs can require more effort than designing a new language.

UML2 includes several *ambiguities*, which further complicate implementing model transformations. For example, designers can specify a loop within a UML2 activity diagram either by iterator nodes or by a control flow pointing backwards. A model transformation has to support all possible cases, which increases its complexity.

8.3. Defining a new MOF-based meta-model

We chose this approach due to the *advantages* to explore modelling and analysing advanced component concepts and performance predictions. UML2 is a general purpose modelling language aiming at providing model constructs for mainly object-oriented systems without focussing on any domain. Opposed to this, the PCM is a domain specific modelling language for component-based software architectures and performance properties. It targets specific developer roles in CBSE providing restricted modelling languages for each of these roles, which intentionally do not inherit UML2's complexity. Model constructs in the PCM are defined unambigiously and are limited to constructs mappable to the performance domain.

The PCM includes several concepts also present in UML2 or the SPT profile, such as interfaces, resource specification, workloads, and performance annotations. Additionally, the PCM includes concepts from CBSE not explicitly present in UML2, such as service effect specifications (Koziolek et al., 2007), a component type hierarchy (Reussner et al., 2006), better support for performance annotations (Becker et al., 2007), and an explicit component context model for expressing a component's QoS in dependence of its environment (Becker et al., 2006). In this paper, we focus on QoS-relevant modelling elements, a specification of the other concepts can be found on the PCM's website (Palladio, 2007).

We implemented the PCM in Ecore from the Eclipse Modelling Framework (EMF), which is an implementation of Essential MOF (EMOF), a subset of MOF. We chose Ecore instead of MOF due to strong tool and community support, which eases implementing graphical editors and analysis tools.

While our approach includes the described advantages, it also bears several *disadvantages*. We do not follow an industry standard, therefore wide-spread use of the language is a long-term goal. We cannot use existing UML tools to create PCM instances and have implemented our own proprietary tools. The learning curve for developers familiar with UML is potentially higher. However, due to the restricted nature of the role-specific modelling languages, the specification is not overly complex. We reused the UML2 graphical notation for PCM instances where possible to increase the willingness of designers to accept and use the new technology.

Finally, existing UML models are not supported by the PCM analysis tools and have to be transformed to PCM instances to carry out performance predictions. While implementing such a transformation is complicated for the reasons described above, it would increase the practical use of the PCM and is subject to our future work.

9. Limitations/assumptions

There are several assumptions and limitations in the current version of the PCM and the accompanying tools. We briefly summarise the most important ones in the following.

Static architecture: The modelled architecture is assumed to be static. This means that neither the connectors change nor that the components can move like agents to different hardware resources.

Limited Connectors: Connector can be attached to a required interface. While it is possible to model such connectors in UML2 their semantics remain unclear.

Abstraction from state: It is assumed that the behaviour of the system is determined by the parameters of the service calls only. No internal state of components or the runtime environment is regarded. We do not consider components at runtime, which may adapt their behaviour to change their QoS properties dynamically. These QoS-aware components are beyond the scope of the PCM.

No memory effects: Components might allocate and free memory during request processing. In the multi-user case, the components may in addition struggle for getting access to the memory bus which is often granted only mutually exclusive. Both effects can have a significant impact on the resulting performance (for measurements see (Happe et al., 2006)). However, our method still disregards them.

Information availability: It is assumed that the necessary model information like service effect specifications and parametric dependencies are available and have been specified by the component developer, software architect, system deployer and domain expert. We also assume that different component developers are able to agree on common parameter abstractions. Future work is directed to retrieve as much information as possible from the automated analysis of component code.

Limited support for concurrency: Quality properties of concurrent systems are hard to predict. Especially on multi-core processor systems several effects like the CPU caches and scheduling strategies lead to differences between an observed system timing behaviour and an appropriate prediction in our experience.

Limited support for modelling the runtime environment: Our resource model assumes that processing resources can be described by a processing rate only. But often more than a single influence factor is important. For example, to characterize modern CPUs by the clock frequency alone, is often not sufficient any more. The CPU architecture, pipelining strategy, or the cache sizes as well as the runtime and middleware platform and their configurations can have a significant influence on the execution time (of an operation) (Liu et al., 2005). The performance prototype is a countermeasure for this limitation as the availability of a precise model of the resource environment is dropped by using the real one.

Mathematical assumptions: Mathematical assumptions and limitations are needed in order to reduce the models' complexity in order to allow the analytical single-user evaluation and to reduce simulation run length and memory consumption. Our current version of the PCM assumes for example stochastical independent resource demands. The only exception to this is the `CollectionIteratorAction` which allows a dependency on parameter characterisations of the current iteration element.

Many of the listed limitations are not specific to the PCM, but can also be found in other performance prediction methods. In our case studies, particularly the limited support for concurrency and multi-processor systems lead to inaccurate predictions in some settings (Happe et al., 2006). In order to determine the most industry-relevant limitations, we are currently conducting several industrial case studies.

10. Case study

In the following, we report on an initial case study to validate predictions made by simulating instances of the PCM. This case study is meant as a proof-of-concept evaluation, not as a sound experimental evaluation of the approach on an industrial sized application, which is planned for the future. We compare predictions based on architectural specifications with measurements made with an implementation of the architecture. The case study involves an online shop, which allows users uploading and downloading music files (Koziolek et al., 2006). In the former paper, we have predicted the performance of this shop with a restricted analytical method instead of the more feature-rich simulation described here. Several parameter dependencies, which will be explained in the following, can be found in the architecture, so we can test the modelling capabilities of our component model.

As we want to support early design decisions with our approach, we modelled and implemented two design alternatives for the online shop. Before the case study, we raised the following questions:

(i) Are the predictions based on our simulation model good enough to support the decision for the design alternative with the actual best performance given a specified usage model?
(ii) Can the errors made by the predictions be quantified and explained? To answer this question, we analysed deviations between predictions and measurement in more detail.

10.1. Architecture

The *architecture* of the "Web Audio Store" (Fig. 30) consists of three tiers (client, application server, database) (Koziolek et al., 2006). Customers interact with the store via web browsers that access the component `WebForm` using digital subscriber lines with a throughput of 128 KBits/s. Several components are located on the application server in the middle tier: The component `WebForm` is connected to the `AudioStore` component, which controls and manages the whole store. It interacts with a user management component and a database adapter that handles the connection to a MySQL server on the database tier. The network between the application server and the database is a dedicated line with a maximum throughput of 512 KBit/s.

As a performance critical *use case*, the response times for uploading music files to the store shall be analysed and improved. It is possible for users to upload multiple files at once to add complete music albums to the store. This usage scenario is described in Fig. 31. In this case, users upload 8–12 files with the probabilities found in the Fig. 31 and files sizes between 3500 and 4500 KBytes. Upon clicking the button "Upload Files" the service `UploadFiles` of the `WebForm` component is invoked, whose behaviour is shown in Fig. 32. This behaviour in turn invokes services from the `AudioStore` component. The parametric dependencies for the loop and the byte size of subsequent input parameters are shown in the figure. For example, the number of loop iterations depends on the number of input files. Other annotations needed for the simulation are omitted in the illustration for brevity.

Because response times of this use case are considered too high, the software architect has come up with a *design alternative*, which is shown within the dashed box in Fig. 30 and which is transparent for the clients. It is proposed to insert an encoder component

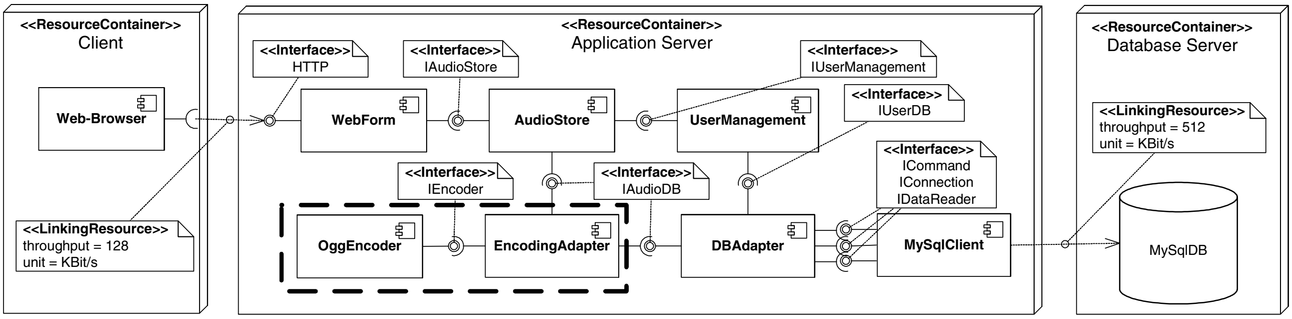


Fig. 30. Web audio store architecture.

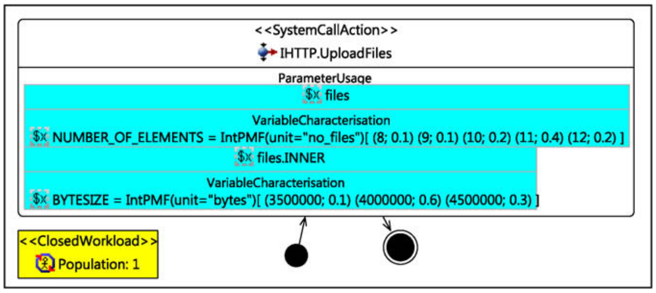

<<SystemCallAction>>
IHTTP.UploadFiles
ParameterUsage
$x files
VariableCharacterisation
$x NUMBER_OF_ELEMENTS = IntPMF(unit="no_files")[(8; 0.1) (9; 0.1) (10; 0.2) (11; 0.4) (12; 0.2)]
$x files.INNER
VariableCharacterisation
$x BYTESIZE = IntPMF(unit="bytes")[(3500000; 0.1) (4000000; 0.6) (4500000; 0.3)]
<<ClosedWorkload>>
Population: 1

Fig. 31. Original design: Usage model.

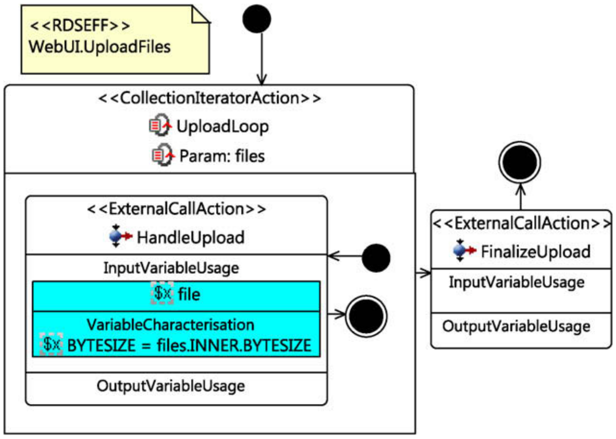


Fig. 32. Original design: SEFF.

(`OggEncoder`) into the architecture using an adapter (`EncodingAdapter`), which implements the `IAudioDB` interface. The SEFFs of these components are illustrated in Fig. 33a and b. The encoder is able to reduce the size of the music files by a factor of approx. 2/3. Thus, the time for using the network connection to the database server can be reduced, because smaller files have to be sent. However, encoding the files is computational intensive and hence, costs an amount of time. With the performance prediction, the trade-off between faster network transfer and re-encoding overhead shall be evaluated.

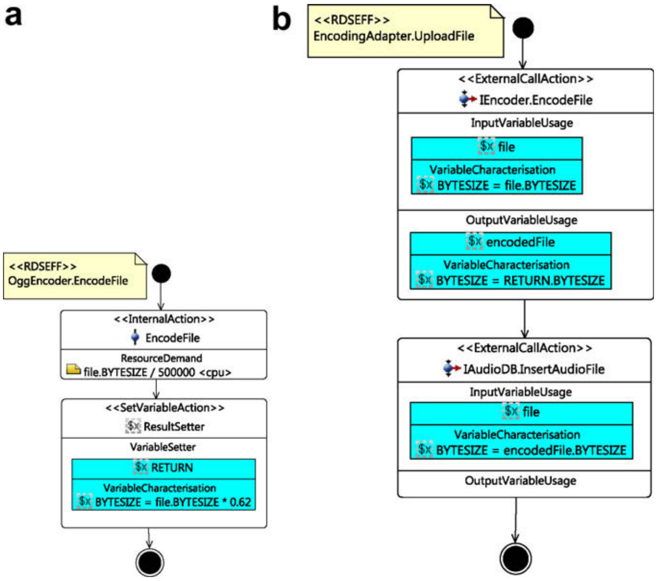


Fig. 33. Design alternative: Encoding adapter.

Table 3
Response time for EncodeFile

Response time (Seconds)	Probability (simulated)	Probability (measured)
5.5–6.5	0.00	0.00
6.5–7.5	0.10	0.18
7.5–8.5	0.58	0.58
8.5–9.5	0.31	0.25
9.5–10.5	0.00	0.00

10.2. Results

First, we modelled and simulated both design alternatives, then we also implemented both alternatives in C# using ASP.NET. Using the workloads from the models, the response times for the described use case of the implementations were measured.

Compared to Koziolek et al. (2006), we only used measured time consumptions in our prediction model for calls to the database, i.e., for submitting uncompressed files. The time consumption for submitting compressed files as well as the encoding of files had been specified parametrically as introduced in the previous section. In so doing, we further weakened the assumptions which we made when we did the predictions in Koziolek et al. (2006).

For the original design without encoder, the simulation results and the measured time consumptions are presented in Fig. 34a and b. To allow a visual comparison of the results, we show both histograms in a single diagram by putting them on top of each other. The simulation results are drawn using a plain line and the measured results as dotted line.

Both functions match to a large extent. Hence, our simulation is capable to predict the response time behaviour of this design alternative based on a system model. In this variant of the architecture, the main influence on the response time is created by the amount of files in a batch upload. As we modelled the distribution of the amount of files according to what we actually used when we measured our system, this result was expected. Additionally, the close match between both functions also results from the use of measured times for the basic functions (like encoding, database storage, etc.). However, if there is a larger gap between the (abstract) architecture and its implementation, the results might not be as good. Ongoing industrial case studies will provide additional insights in the question how to choose the right abstraction.

For the design alternative with the encoder, we first compared the simulation results of the call to `EncodeFile` with the measured values as we used a parametric dependency on the bytesize of the file to encode. The dependency was derived by a rough guess looking at a few sample encoder runs. The result is depicted in Table 3.

It can be seen that our estimated dependency is not exactly matching, but still quite good. Using the estimated times of `EncodeFile` and the compression rate of 2/3 we simulated the whole system. The results are shown as response time histogram in Fig. 35a. The cumulative distribution function (CDF) of the results are shown in Fig. 35b.

Although we used the mentioned approximations of the parametric dependencies, the measured and the simulation results still match quite good. Moreover, we are able to see that the design alternative with the encoder is approximately 200 seconds faster. Our result favoured the design alternative with the encoder, which is indeed the faster one as validated by the measurements.

To answer the question whether observable prediction errors can be explained, we take a closer look at Fig. 35a which has minor prediction errors that can be explained by the guessed dependency for `EncodeFile` and the database connection. Both are not as accurate as they could be. A linear regression based on some mea-

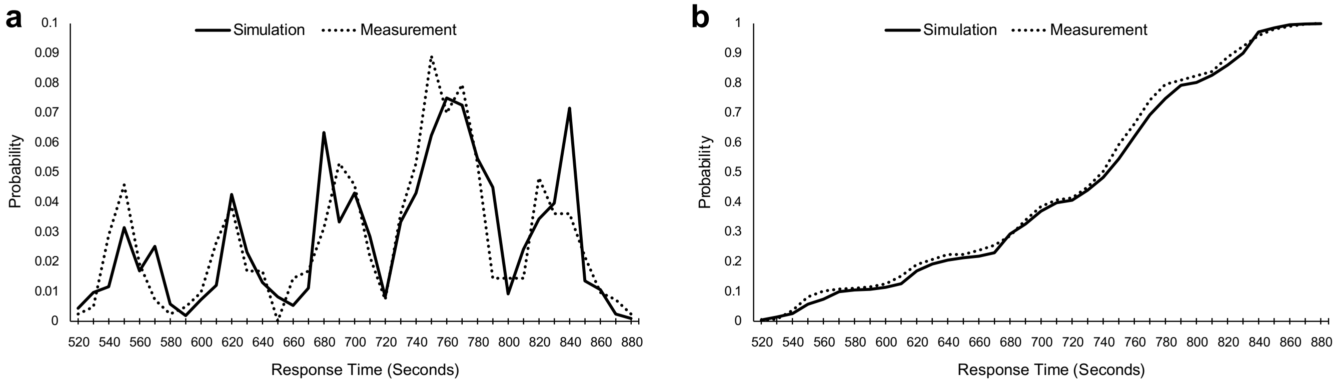


Fig. 34. Measurement and simulation result of response time of UploadFiles (architecture without encoder).

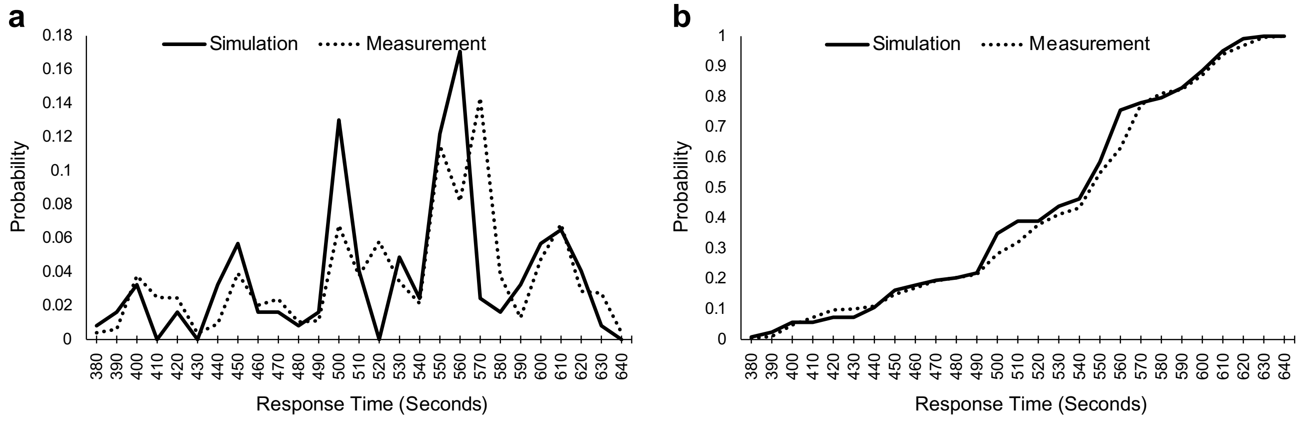


Fig. 35. Measurement and simulation result of response time of UploadFiles (architecture with encoder).

sured time consumptions for different bytesizes could help to build a better estimation of the actual dependency. However, software architects often have to rely on estimates in practice which is why we also used the estimates.

11. Conclusions

This paper presents a meta-model designed to support the prediction of extra-functional properties of component-based software architectures. Different possible usage contexts of a component are supported by this model. Additionally, parametric context dependencies to system resources and dependencies to parameter usages can be modelled. We present tool support for modelling and model transformations which transform instances of the PCM in different types of prediction models. The paper presents technical details on a simulation framework for a model-driven analysis of PCM instances and validates the soundness of the predictions in a case study.

The presented method is designed to support early design time quality evaluations of component-based software architectures. Based on models, a software architect can evaluate the quality of the modelled system. The evaluation of design alternatives can be performed by changing the input models and re-running the transformations. The focus on system models enables a quick feedback cycle by the use of ideas from model-driven development.

A new hybrid (analytical and simulation) method based on an extended stochastic process algebra is currently under development which is supposed to speed up the simulation runs by evaluating sub-processes using results from our analytical single-user method in advance.

Simulation based predictions for concurrent system usage are still limited due to the state and memory usage assumptions. However, for systems with a lot of concurrency the simulation can serve as a basis for experiments and comparisons with running systems in order to improve the model to increase prediction precision.

Additionally, we are currently conducting a student experiment to gain insights in the applicability and understandability of our models and tools by common software developers. Special attention is payed to the question whether annotating models with performance and usage information is feasible.

Further enhancements to our toolset are planned for improved evaluation of prediction results. Performance critical `InternalAction`s could be highlighted in the editors. Support for automated generation of design alternatives to cope with performance issues is still an open issue.

Finally, our first results in recovering `RDSEFF`s from existing Java source code will be included into our toolset. The availability of the reverse engineering facility allows for creating PCM model instances for legacy components.

Acknowledgement

We thank all members of the Chair “Software Design and Quality” from the University of Karlsruhe for fruitful discussions and extensive proof-reading.

References

Arief, L.B., Speirs, N.A., 2000. A UML tool for an automatic generation of simulation programs. In: Proceedings of the Second International Workshop on Software and Performance. ACM Press.

Balsamo, S., Di Marco, A., Inverardi, P., Simeoni, M., 2004. Model-based performance prediction in software development: a survey. IEEE Transactions on Software Engineering 30 (5), 295–310.

Balsamo, S., Marzolla, M., 2003. A Simulation-Based Approach to Software Performance Modeling. In: Proceedings of the 9th European Software Engineering Conference held jointly with 11th ACM SIGSOFT international symposium on Foundations of Software Engineering. ACM Press.

Bause, F., Kritzinger, P.S., 1996. Stochastic Petri Nets: an Introduction to the Theory. Vieweg-Verlag.

Becker, S., Grunske, L., Mirandola, R., Overhage, S., 2006. Performance Prediction of Component-Based Systems: A Survey from an Engineering Perspective. In: Reussner, R., Stafford, J., Szyperski, C. (Eds.), Architecting Systems with Trustworthy Components, LNCS, vol. 3938. Springer, pp. 169–192.

Becker, S., Happe, J., Koziolek, H., 2006. Putting Components into Context - Supporting QoS-Predictions with an explicit Context Model. In: Reussner, R., Szyperski, C., Weck W. (Eds.), Proceedings of the Eleventh International Workshop on Component-Oriented Programming (WCOP'06).

Becker, S., Koziolek, H., Reussner, R., 2007. Model-based performance prediction with the Palladio component model. In: Proceedings of the 6th International Workshop on Software and Performance (WOSP2007). ACM Sigsoft.

Bertolino, A., Mirandola, R., 2004. CB-SPE Tool: Putting component-based performance engineering into practice. In: Crnkovic, I., Stafford, J.A., Schmidt, H.W., Wallnau, K.C. (Eds.), Proc. 7th International Symposium on Component-Based Software Engineering (CBSE 2004), Edinburgh, UK, Lecture Notes in Computer Science, vol. 3054. Springer-Verlag, Berlin, Germany.

Bolch, G., Greiner, S., de Meer, H., Trivedi, K.S., 1998. Queueing Networks and Markov Chains. John Wiley and Sons Inc..

Bondarev, E., de With, P., Chaudron, M., Musken, J., 2005. Modelling of Input-Parameter Dependency for Performance Predictions of Component-Based Embedded Systems. In: Proceedings of the 31th EUROMICRO Conference (EUROMICRO'05.

Bondarev, E., de With, P.H.N., Chaudron, M., 2004. Predicting Real-Time Properties of Component-Based Applications. In: Proceedings of RTCSA.

Chang, X., 1999. Network simulations with OPNET. In: WSC'99: Proceedings of the 31st Conference on Winter Simulation. ACM Press, New York, NY, USA.

Cheesman, J., Daniels, J., 2000. UML Components: A Simple Process for Specifying Component-based Software. Addison-Wesley, Reading, MA, USA.

Chen, S., Liu, Y., Gorton, I., Liu, A., 2005. Performance prediction of component-based applications. Journal of Systems and Software 74 (1), 35–43.

Cortellessa, V., 2005. How far are we from the definition of a common software performance ontology? In: WOSP'05: Proceedings of the 5th International Workshop on Software and Performance. ACM Press, New York, NY, USA.

Cortellessa, V., Pierini, P., Rossi, D., 2007. Integrating software models and platform models for performance analysis. IEEE Transactions on Software Engineering 33 (6), 385–401.

CSIM, Performance Simulator (May 2007) URL http://www.csim.com/.

de Miguel, M., Lambolais, T., Hannouz, M., Betge-Brezetz, S., Piekarec, S., 2000. UML extensions for the specification and evaluation of latency constraints in architectural models. In: WOSP'00: Proceedings of the 2nd International Workshop on Software and Performance. ACM Press, New York, NY, USA.

Denaro, G., Polini, A., Emmerich, W., 2004. Early performance testing of distributed software applications. SIGSOFT Software Engineering Notes, vol. 29. ACM Press, New York, NY, USA.

Denning, P.J., Buzen, J.P., 1978. The Operational Analysis of Queueing Network Models. ACM Computing Survey 10 (3), 225–261.

The DESMO-J Homepage, last retrieved 2008-01-06 (2007) URL <http://asi-www.informatik.uni-hamburg.de/desmoj/>.

Eskenazi, E., Fioukov, A., Hammer, D., 2004. Performance prediction for component compositions. In: Proceedings of the 7th International Symposium on Component-based Software Engineering (CBSE7). Lecture Notes in Computer Science, vol. 3054. Springer-Verlag, Berlin, Germany.

Goebel, S., Pohl, C., Roettger, S., Zschaler, S., 2004. The COMQUAD component model: enabling dynamic selection of implementations by weaving non-functional aspects. In: AOSD'04: Proceedings of the 3rd International Conference on Aspect-oriented Software Development. ACM Press, New York, NY, USA.

Gomaa, H., Menasce, D.A., 2001. Performance engineering of component-based distributed software systems. In: Performance Engineering, State of the Art and Current Trends. Springer-Verlag, London, UK.

Grassi, V., Mirandola, R., Sabetta, A., 2007. Filling the gap between design and performance/reliability models of component-based systems: A model-driven approach. Journal on Systems and Software 80 (4), 528–558.

Hamlet, D., Mason, D., Woit, D., 2004. Component-based software development: case studies. Series on Component-Based Software Development, chap. Properties of Software Systems Synthesized from Components, vol. 1. World Scientific Publishing Company.

Happe, J., Koziolek, H., Reussner, R., 2006. Parametric Performance Contracts for Software Components with Concurrent Behaviour. In: de Boer, F.S., Mencl V. (Eds.), Proceedings of the 3rd International Workshop on Formal Aspects of Component Software (FACS06), Prague, Czech Republic, Electronical Notes in Computer Science.

Heineman, G.T., Councill, W.T. (Eds.), 2001. Component-Based Software Engineering. Addison Wesley.

Henderson-Sellers, B., 2005. UML – the good, the bad or the ugly? perspectives from a panel of experts. Software and System Modeling 4 (1), 4–13.

Hermanns, H., Herzog, U., Katoen, J.-P., 2002. Process algebra for performance evaluation. Theoretical Computer Science 274 (1–2), 43–87.

Hissam, S.A., Moreno, G.A., Stafford, J.A., Wallnau, K.C., 2002. Packaging Predictable Assembly. In: Bishop, J.M. (Ed.), Component Deployment, IFIP/ACM Working Conference, CD 2002, Berlin, Germany, June 20–21, 2002, Proceedings, vol. 2370 of Lecture Notes in Computer Science, Springer.

Jain, R., 1991. The Art of Computer Systems Performance Analysis: Techniques for Experimental Design, Measurement, Simulation, and Modeling. Wiley.

Kleinrock, L., 1975. Queuing Systems, Vol. 1: Theory. Wiley and Sons, New York, NY, USA.

Kounev, S., 2006. Performance modeling and evaluation of distributed component-based systems using queueing petri nets. IEEE Transactions on Software Engineering 32 (7), 486–502.

Koziolek, H., Becker, S., Happe, J., 2007. Predicting the performance of component-based software architectures with different usage profiles. In: Proceedings 3rd International Conference on the Quality of Software Architectures (QoSA'07). LNCS, vol. 4880. Springer.

Koziolek, H., Happe, J., 2006. A Quality of Service Driven Development Process Model for Component-based Software Systems. In: Gorton, I., Heineman, G.T., Crnkovic, I., Schmidt, H.W., Stafford, J.A., Szyperski, C.A., Wallnau, K.C. (Eds.), Component-Based Software Engineering, Lecture Notes in Computer Science, vol. 4063. Springer-Verlag, Berlin, Germany http://dx.doi.org/10.1007/11783565_25.

Koziolek, H., Happe, J., Becker, S., 2006. Parameter dependent performance specification of software components. In: Proceedings of the Second International Conference on Quality of Software Architectures (QoSA2006). Lecture Notes in Computer Science, vol. 4214. Springer-Verlag, Berlin, Germany.

Lau, K.-K., Wang, Z., 2005. A Taxonomy of Software Component Models. In: Proceedings of the 31st EUROMICRO Conference. IEEE Computer Society Press.

Lazowska, E., Zahorjan, J., Graham, G.S., Sevcik, K.C., 1984. Quantitative System Performance – Computer System Analysis Using Queueing Network Models. Prentice-Hall.

L'Ecuyer, P., Buist, E., 2005. Simulation in Java with SSJ, in: WSC'05: Proceedings of the 37th Conference on Winter simulation, Winter Simulation Conference.

Liu, Y., Fekete, A., Gorton, I., 2005. Design-level performance prediction of component-based applications. IEEE Transactions on Software Engineering 31 (11), 928–941.

Martens, A., 2007. Empirical Validation of the Model-driven Performance Prediction Approach Palladio, Master's thesis, Universität Oldenburg (Nov. 2007). URL <http://sdq.ipd.uka.de/diploma_theses_study_theses/compl%eted_theses>.

McIlroy, M.D., 1968. "Mass Produced" Software Components. In: Naur, P., Randell, B. (Eds.), Software Engineering, Scientific Affairs Division, NATO, Brussels, 1969, Report of a Conference Sponsored by the NATO Science Committee, Garmisch, Germany, 7th to 11th October 1968.

Medvidovic, N., Rosenblum, D.S., Redmiles, D.F., Robbins, J.E., 2002. Modeling software architectures in the Unified Modeling Language. ACM Transactions on Software Engineering Methodology 11 (1), 2–57.

Medvidovic, N., Taylor, R.N., 2000. A classification and comparison framework for software architecture description languages. IEEE Transactions on Software Engineering 26 (1), 70–93.

Menasce, D.A., Almeida, V.A.F., Dowdy, L.W., 2004. Performance by Design. Prentice Hall.

Object Management Group (OMG), 2005. UML Profile for Schedulability, Performance and Time (January 2005). URL <http://www.omg.org/cgi-bin/doc?formal/2005-01-02>.

Object Management Group (OMG), 2006. UML Profile for Modeling and Analysis of Real-Time and Embedded systems (MARTE) RFP (realtime/05-02-06). URL <http://www.omg.org/cgi-bin/doc?realtime/2005-2-6>.

Object Management Group (OMG), 2007. Meta Object Facility (MOF) 2.0 Query/View/Transformation Specification (ptc/07-07-07). URL <http://www.omg.org/docs/ptc/07-07-07.pdf>.

openArchitectureWare (oAW), 2007. openArchitectureWare (oAW) Generator Framework, last retrieved 2008-01-06. URL <http://www.openarchitectureware.org>.

Palladio, 2007. The Palladio Component Model (Download and Documentation), <http://sdqweb.ipd.uka.de/wiki/Palladio_Component_Model> (May 2007).

Petriu, D.B., Woodside, M., 2004. A Metamodel for Generating Performance Models from UML Designs. In: Baar, T., Strohmeier, A., Moreira, A., Mellor, S.J. (Eds.), UML 2004 – The Unified Modeling Language. Model Languages and Applications. 7th International Conference, Lisbon, Portugal, October 11–15, 2004, Proceedings, vol. 3273 of LNCS, Springer.

Reussner, R.H., 2003. Contracts and Quality Attributes of Software Components. In: Weck, W., Bosch, J., Szyperski, C. (Eds.), Proceedings of the Eighth International Workshop on Component-Oriented Programming (WCOP'03).

Reussner, R.H., Becker, S., Happe, J., Koziolek, H., Krogmann, K., Kuperberg, M., 2006. The Palladio Component Model, Technical Report. Universitaet Karlsruhe (TH).

Reussner, R.H., Schmidt, H.W., Poernomo, I., 2003. Reliability Prediction for Component-Based Software Architectures. Journal of Systems and Software – Special Issue of Software Architecture – Engineering Quality Attributes 66 (3), 241–252.

Rolia, J.A., Sevcik, K.C., 1995. The Method of Layers. IEEE Transactions on Software Engineering 21 (8), 689–700.

Sitaraman, M., Kuczycki, G., Krone, J., Ogden, W.F., Reddy, A., 2001. Performance Specification of Software Components. In: Proceedings of the 2001 symposium on Software reusability: putting software reuse in context. ACM Press.

Smith, C.U., 1990. Performance Engineering of Software Systems. Addison-Wesley, Reading, MA, USA.

Smith, C.U., Llado, C.M., Cortellessa, V., Di Marco, A., Williams, L.G., 2005. 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. ACM Press, New York, NY, USA.

Smith, C.U., Williams, L.G., 2002. Performance Solutions: A Practical Guide to Creating Responsive, Scalable Software. Addison-Wesley.

Szyperski, C., Gruntz, D., Murer, S., 2002. Component Software: Beyond Object-Oriented Programming, second ed. ACM Press and Addison-Wesley, New York, NY.

Trivedi, K.S., 1982. Probability and Statistics with Reliability, Queuing and Computer Science Applications. Prentice Hall, Englewood Cliffs, NJ, USA.

Völter, M., Stahl, T., 2006. Model-Driven Software Development. Wiley and Sons, New York, NY, USA.

Woodside, C.M., Neilson, J.E., Petriu, D.C., Majumdar, S., 1995. The Stochastic Rendezvous Network Model for Performance of Synchronous Client-Server-like Distributed Software. IEEE Transactions on Computers 44 (1), 20–34.

Wu, X., Woodside, M., 2004. Performance Modeling from Software Components. SIGSOFT Software Engineering Notes 29 (1), 290–301.

Yacoub, S.M., 2002. Performance analysis of component-based applications. In: SPLC 2: Proceedings of the Second International Conference on Software Product Lines. Springer-Verlag, London, UK.

Steffen Becker is Department Head at the IT Research Centre in Karlsruhe (FZI) in the division for Software Engineering since January 2008. Before he graduated from the University of Oldenburg with a PhD in computer science. From July 2003 he was a member of the young investigators excellence program "Palladio" of the National German Research Foundation. He got his diploma in business administration and computer science combined in 2003 from the Technical University of Darmstadt. He participated in the orginisation of several conferences and workshops, including QoSA and WCAT. His research interests include model-driven software development, software architectures, and model-driven quality predictions.

Heiko Koziolek is working for the Industrial Software Technologies Group of ABB Research in Ladenburg, Germany. He received his PhD degree in computer science from the University of Oldenburg, Germany, in 2008. From 2005 to 2008 Heiko was a member of the graduate school "TrustSoft" and held a PhD-scholarship from the German Research Foundation. He has worked as an intern for IBM and sd&m. His research interests include performance engineering, software architecture evaluation, model-driven software development, and empirical software engineering.

Ralf Reussner is full professor for software engineering at the University of Karlsruhe and holds the Chair for Software-Design and -Quality since 2006. His research interests include software components, software architecture and model-based quality prediction. Ralf graduated from University of Karlsruhe with a PhD in 2001 and was with the DSTC Pty Ltd, Melbourne, Australia. From 2003 till 2006 he held a Juniorprofessorship for Software Engineering at the University of Oldenburg, Germany. Ralf is organiser of various conferences and workshops, including QoSA and WCOP. As Director of Software Engineering at the IT Research Centre in Karlsruhe (FZI) he consults various industrial partners.