Karlsruhe Reports in Informatics 2011,14
Edited by Karlsruhe Institute of Technology,
Faculty of Informatics
ISSN 2190-4782

The Palladio Component Model

Ralf Reussner, Steffen Becker, Erik Burger, Jens Happe,
Michael Hauck, Anne Koziolek, Heiko Koziolek,
Klaus Krogmann, Michael Kuperberg

22 March 2011
Chair for Software Design & Quality (SDQ)
Karlsruhe Institute of Technology, Germany

2011

KIT – University of the State of Baden-Wuerttemberg and National Research Center of the Helmholtz Association

Fakultät für **Informatik**

Contents

List of Figures

Chapter 1

Introduction

1.1 Motivation

This report introduces the Palladio Component Model (PCM), a novel software component model for business information systems, which is specifically tuned to enable model-driven quality-of-service (QoS, i.e., performance and reliability) predictions (based on work previously published in [1, 2, 3, 4, 5]). The PCM's goal is to assess the expected response times, throughput, and resource utilization of component-based software architectures during early development stages. This shall avoid costly redesigns, which might occur after a poorly designed architecture has been implemented. Software architects should be enabled to analyse different architectural design alternatives and to support their design decisions with quantitative results from performance or reliability analysis tools.

Component-based software engineering (CBSE) [6] promises many advantages over object-oriented or procedural development approaches. Besides increased reusability, better preparation for evolution, higher quality due to increased testing, and shorter time-to-market, CBSE potentially offers better predictability for the properties of architectures, because individual components should be provided with more detailed specifications. A large number of component models has been designed for different purposes. Component models used in the industry today (COM+/.NET, J2EE/EJB, CCM, etc.) do not offer capabilities for predicting QoS attributes. Component models from academia [7] have been designed to support purposes like runtime configuration, protocol checking, mobile device assessment etc. Some of them deal with QoS predictions (e.g., ROBOCOP, KLAPER, CB-SPE, PACC, MARTE etc.), but often have a different notion of software components.

Model-based QoS-prediction approaches for determining the performance and reliability of software systems have been researched for decades, but are still hardly used in practice. A survey by [8] classifies recent performance prediction approaches, and the overview by [9] includes a large number of reliability models. These approaches mostly target monolithic systems and are usually not sufficiently tuned for component-based systems. Specifying QoS properties of independently deployable software components is difficult, because component developers cannot know on what kind of machine their code is used, what parameters will be supplied to the component's provided services, and how the components required services will react.

Two key features of the PCM are i) the parameterised component QoS specification and ii) the developer role concept. Concerning i), the PCM is based on the component definition introduced by [6]. Software components are black box entities with contractually specified interfaces. They encapsulate their contents and are a unit of independent deployment. Most importantly, components can be composed with other components via their interfaces. The PCM offers a special QoS-specification for software components, which is parameterised over environmental influences, which should be unknown to component developers during design and implementation.

Concerning ii), the domain-specific language of the PCM is aligned to the different roles involved in component based development. Component developers specify models of individual components, which are composed by software architects to architectures. Deployers can model the hardware/VM/OS-environment of the architecture, and domain experts are enabled to supply a description of the user's behaviour, which is necessary for QoS predictions. A QoS-driven development process model supports the roles in combining their models.

1.2 Overview

This report is structured as follows:

- **Chapter 2:** lays the foundation to understand the concepts of the PCM. First, the QoS-driven development process model targeted by the PCM is introduced (2.1). Basic principles of interfaces and composition are highlighted. (2.2). Parametric contracts enable adapting the pre-/postconditions of components (2.3). As QoS of component depends on the context a component is executed in, the PCM introduces a special context concept (2.4). To specify resource demands, random variables can be used in the PCM (2.5).
- **Chapter 3:** presents the concepts of the PCM and is structured after the roles involved in modeling. Component developers specify interfaces, services, components, and QoS properties (3.1). Software architects use the specifications of component developers to build architectures (3.2). Deployers model the resource environment of an architecture (3.3). Domain experts provide information about the user behaviour (3.4). QoS experts collect the information from the different roles, use prediction tools and pre-interpret the results (3.5).
- **Chapter 4:** contains the generated technical reference of the PCM metamodel.
- **Chapter 5:** discusses the PCM, describes related work, open issues, as well as limitations and assumptions present in the PCM.

Chapter 2

Foundations

2.1 Component-based Development Process

2.1.1 Motivation

Component-based software development follows a different process than classical procedural or object-oriented development [6]. The task of developing software artefacts is split between the role of the component developer, who develops individual components, and the software architect, who assembles those components to form an application. Further roles are involved in specifying requirements and defining the resource environment.

For using the PCM, a specific development process with specific developer roles is envisioned [2], which builds on an existing component-based development process introduced by Cheeseman and Daniels [10], which was in turn based on the Rational Unified Process (RUP).

Early QoS analyses of a component-based architectures depend on information about its usage profile and resource environment. This information might not be available directly from software architects or component developers. Thus, further *developer roles*, such as deployers, domain experts and QoS analysts are needed for the specification and QoS analysis of a component-based architectures. These developer roles and their tasks are described in Section 2.1.2. For the PCM, a domain specific modeling language has been created for each of these roles. These modeling languages will be described in detail in Chapter 3.

The PCM process model extends the *process model* by Cheeseman and Daniels (Section 2.1.3). Section 2.1.4 elaborates on the specification workflow and illustrates the interdependencies between component developer and software architect. The PCM process model additionally contains a workflow ”QoS-Analysis” (Section 2.1.5), in which all of the developer roles interact to predict the performance or reliability of the architecture.

The development process introduced in the following is generic, so that it could be followed by other model-based QoS prediction approaches for component systems [11] as well. It is furthermore not restricted to a specific QoS property like performance, but can also be used for reliability, availability, security, safety, etc. The process model reflects our vision of software development including early QoS analyses. Its applicability in practice remains to be validated. Some discussion points about the process model as well as related work are summed up in Section 2.1.6.

2.1.2 Roles in Component-based Development

Figure 2.1: Concept of Roles.

Before introducing the individual roles of the component-based development process, we describe the general concept of roles in software development. Figure 2.1 illustrates the relation of persons, roles, and tasks. A *role* groups a set of tasks that have an overall purpose and each *task* is associated to exactly one role. For example, the role component developer performs tasks like component implementation and component specification. A role can be adopted by multiple *persons*, e.g. there can be multiple persons who are component developers involved in the process. On the other hand, it is also possible for a person to adopt multiple roles. For instance, some component developers might also play the role of software architects, who are responsible for designing the software architecture. The relation of persons and roles is an important concept and has to be considered when reading the following descriptions of roles.

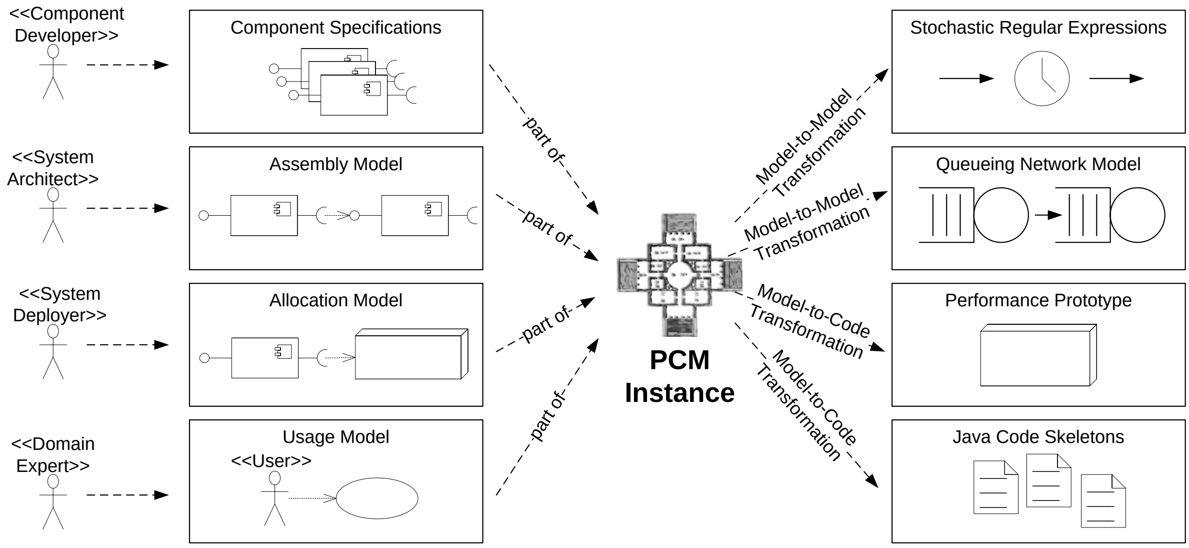


Figure 2.2: Developer Roles in the Palladio Process Model

Since we want to evaluate QoS attributes during early development stages, we need additional information about the internal component structure, the usage model, and the execution environment. These cannot be provided by a single person (e.g., the software architect) involved in the development process, since the required knowledge is spread among different persons with different development roles. Therefore, a QoS analyst requires support of component developers, software architects, domain experts, and deployers to analyse the QoS attributes of a software architecture (cf. Fig. 2.2).

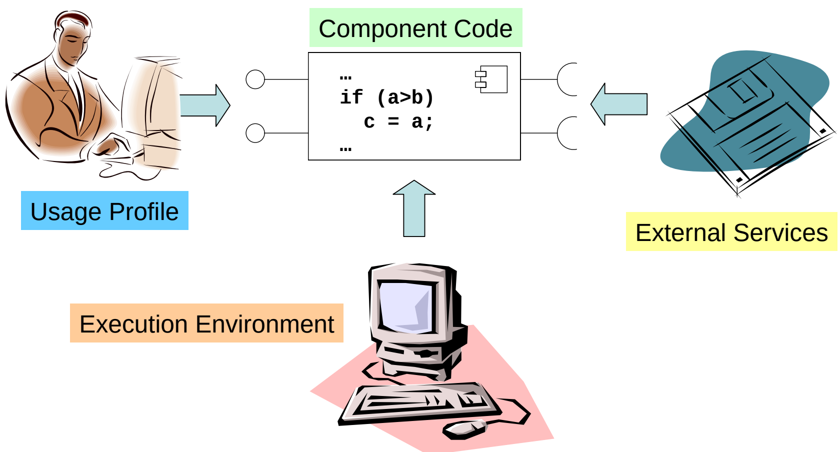


Figure 2.3: Influences on the QoS properties of a Software Component

Component developers are responsible for the specification and implementation of components. They develop components for a market as well as per request. To enable QoS analyses, they need to specify the QoS properties of their components without knowing a) to which other components they are connected, b) on which hardware/software platform they are executed, and c) which parameters are used when calling their services (cf. Figure 2.3). Only such a specification enables independent third party analyses. In the PCM, component developers use service effect specifcations (SEFF, cf. Section 3.1.4) to characterise the QoS properties of their components.

Software architects lead the development process for a complete component-based application. They design the software architecture and delegate tasks to other involved roles. For the design, the planned application specification is decomposed into component specifications. Existing component specifications can be selected from repositories to plan the integration of existing components into a software architecture. If no existing specification matches the requirements for a planned component, a new component has to be specified abstractly. The software architect can delegate this task to component developers. Additionally, the software architect specifies component connections thereby creating an *assembly model* (cf. Section 3.2.2). After finishing the design, software architects are responsible for provisioning components (involving make-or-buy decisions), assembling component implementations, and directing the tests of the complete application.

Deployers specify the resources, on which the planned application shall be deployed. Resources can be hardware resources, such as CPUs, storage devices, network connections etc., as well as software resources, such as thread pools, semaphores or database connection. The result of this task is a so-called *resource environment specification* (cf. Section 3.3.4). With this information, the platform independent resource demands from the component specifications can be converted into timing values, which are needed for QoS analyes. For example, a component developer may have specified that a certain action of a component service takes 1000 CPU cycles. The resource environment specification of the deployer provides the information how many cycles the CPU executing the component processes per second. Both information together yield the execution time of the action. Besides resource specification, deployers allocate components to resources. In the PCM, this step can also be done during design by creating a so-called *allocation model* (cf. Section 3.3.5). Later in the development process, during the deployment stage, deployers can be responsible for the installation, configuration, and start up of the application.

Domain experts participate in requirement analysis, since they have special knowledge of the business domain. They are familiar with the users' work habits and are therefore responsible for analysing and describing the user behaviour. This includes specifying workloads with user arrival rates or user populations and think times. In some cases, these values are already part of the requirement documents. If method parameter values have an influence on the QoS of the system, the domain experts may characterise these values. The outcome of the domain experts' specification task is a so-called *usage model* (cf. Section 3.4.2).

QoS analysts collect and integrate information from the other roles, extract QoS information from the requirements (e.g., maximal response times for use cases), and perform QoS analyses by using mathematical models or simulation. Furthermore, QoS analysts estimate missing values that are not provided by the other roles. For example, in case of an incomplete component specification, the resource demand of this component has to be estimated. Finally, they assist the software architects to interpret the results of the QoS analyses.

2.1.3 Development Process Model

In the following, the roles described in the former section are integrated into a development process model featuring QoS analysis. We focus on the *development process* that is concerned with creating a working system from requirements and neglect the concurrent *management process* that is concerned with time planning and controlling. We base our model on the UML-centric development process model described by Cheeseman and Daniels [10], which is itself based on the Rational Unified Process (RUP).

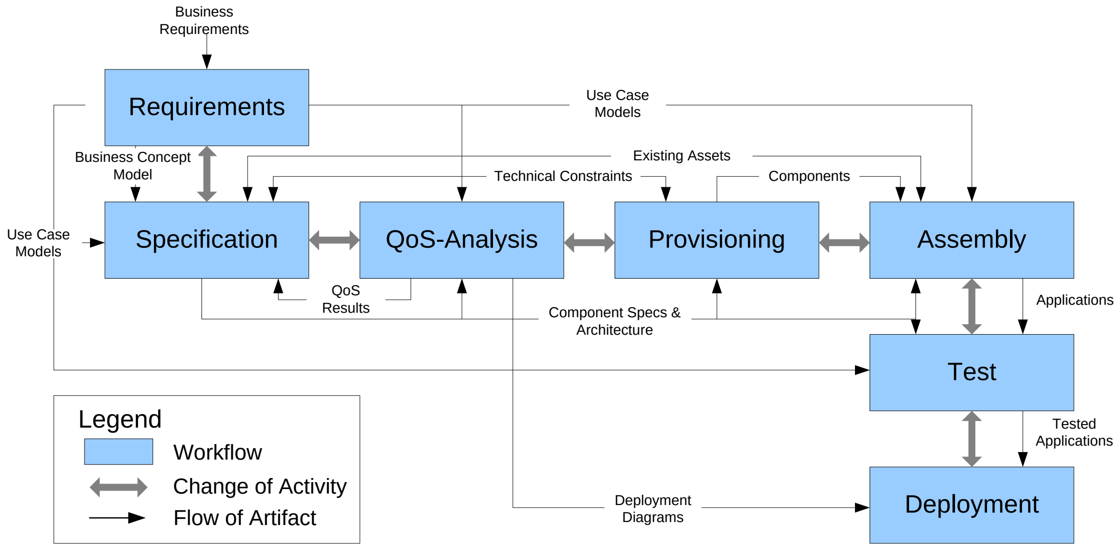


Figure 2.4: QoS Driven Process Model: Overview

The main process is illustrated in Figure 2.4. Each box represents a workflow. Thick arrows between boxes represent a change of activity, while the thin arrows characterise the flow of artifacts between the workflows. The workflows do not have to be traversed sequentially (i.e., no waterfall model). Backward steps into former workflows are allowed. The model also allows an incremental or iterative development based on prototypes.

The workflows *requirements*, *provisioning*, *assembly*, *test*, and *deployment* have mainly been inherited from the original model and will briefly be described in the following. The workflow ”specification” has been slightly modified to explicitly include the interaction between component developer and software architect and the specification of extra-functional properties. The workflow ”QoS Analysis” has been added to the model and will be described in detail below.

Requirements The business requirements coming from customers are formalised and analysed during this workflow. It produces a business concept model and a use case model. The former is a conceptual model of the business domain and creates a common vocabulary between customers and developers. It is, however, not relevant for QoS Analysis. The latter describes the interaction between users (or other external actors) with the system. It establishes the system boundaries and a set of use cases that define the functional requirements.

Specification Business concept model and use case model are input from the requirements to this workflow. Additionally, technical constraints, which might have been revealed during provisioning, and QoS metrics from already performed QoS predictions can be input to the specification workflow after initial iterations of the process model. During specification, the component-based software architecture is designed. Components are identified, specified, and their interaction is defined. The software architect usually interacts with component developers during this workflow. More detail about this workflow is provided in Section 2.1.4. The output artifacts of this workflow are complete component specifications (in PCM also extra-functional specifications) and the component architecture (called *assembly model* in the PCM).

QoS Analysis Component specifications, the architecture, and use case models are input to the QoS analysis workflow. During this workflow, deployers provide models of the resource environment of the architecture, which contain specifications of extra-functional properties (Section 3.3). The domain expert takes the use case models, refines them, and adds QoS-relevant information, thereby creating a PCM usage model (Section 3.4). Finally, the QoS-Analyst a) combines all of the models, b) estimates missing values, c) checks the models' validity, d) feeds them into QoS predictions tools, and e) prepares a pre-evaluation of their predictions, which is targeted at supporting the design decisions of the software architect. More detail about the QoS analysis workflow follows in Section 2.1.5. Outputs of the QoS analysis are pre-evaluated results for QoS metrics, which can be used during specification to adjust the architecture, and deployment diagrams that can be used during deployment.

Provisioning Compared to classical development processes the provisioning workflow resembles the classical implementation workflow. However, one of the assets of component-based development is reuse, i.e. the incorporation of components developed by third parties. During the provisioning workflow "make-or-buy" decisions are made for individual components. Components that cannot be purchased from third-parties have to be implemented according to the specifications from the corresponding workflow. Consequently, the provisioning workflow receives the component specifications and architecture as well as technical constraints as inputs. The outputs of this workflow are implemented software components.

Assembly Components from the provisioning workflow are used in the assembly workflow. Additionally, this workflow builds up on the component architecture and the use case model. The components are assembled according to the assembly model during this workflow. This might involve configuring them for specific component containers or frameworks. Furthermore, for integrating legacy components it might be necessary to write adapters to bridge unfitting interfaces. The assembled components and the complete application code are the outputs of this workflow.

Test The complete component-based application is tested according to the use case models in this workflow in a test environment. It also includes measuring the actual extra-functional properties of the application and their comparison with the predicted values. Once the functional properties have been tested and the extra-functional properties are satisfiable in the test environment the application is ready for deployment in the actual customer environment.

Deployment During deployment, the tested application is installed in its actual customer environment. The term deployment is also used to denote the process of putting components into component containers, but here the term refers to a broader task. Besides the installation, it might be necessary to adopt the resource environment at the customer's facilities or to instruct future users of the system. For the mapping of components to hardware resources, the deployment diagrams from the QoS analysis workflow can be used.

2.1.4 Specification Workflow

The specification workflow (see Figure 2.5, right column) is carried out by software architects. The workflows of the software architect and the component developers influence each other. Existing components (e.g., from a repository) may have an impact on the inner *component identification* and *component specification* workflow, as the software architect can reuse existing interfaces and specifications. Vice versa, components newly specified by the software architect serve as input for the component requirements analysis of component developers, who design and implement new components.

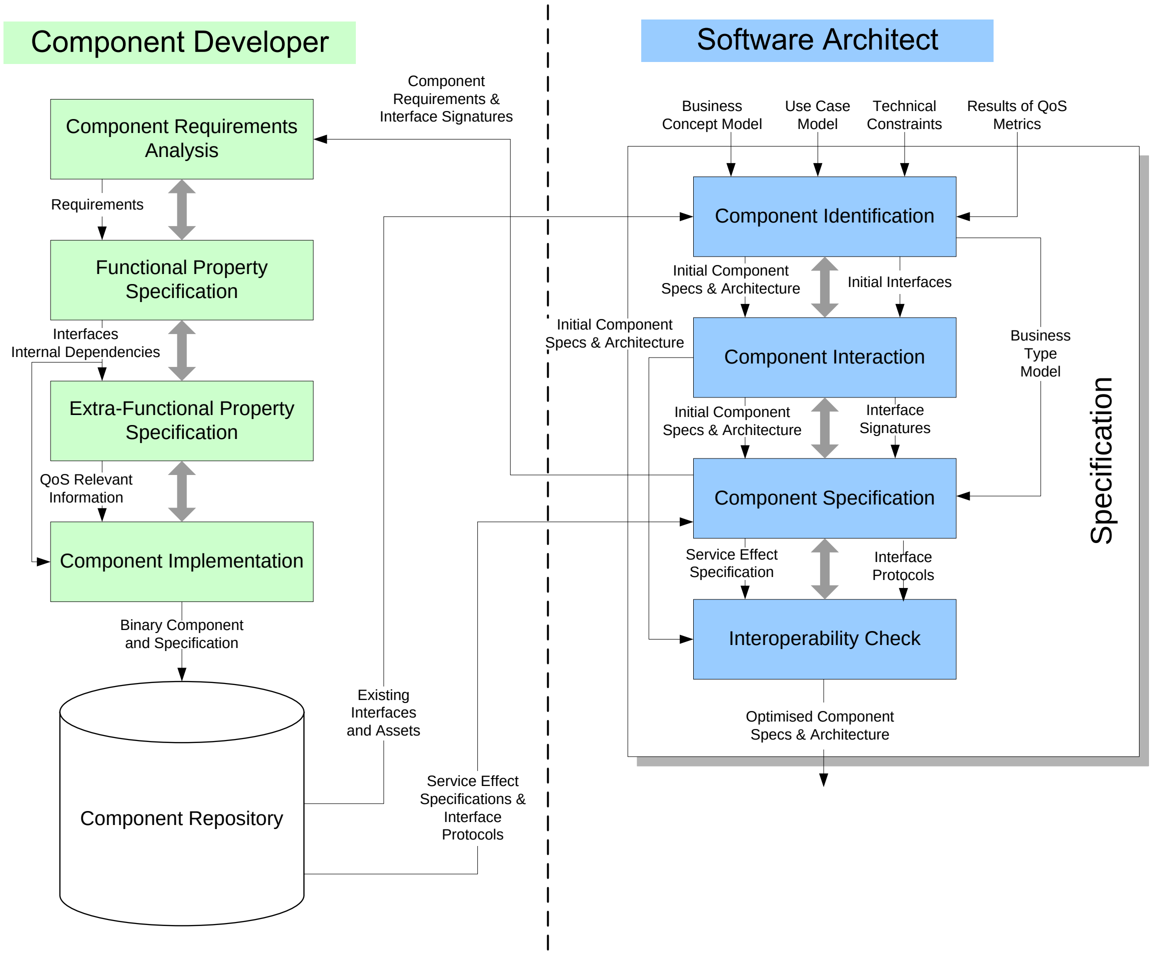


Figure 2.5: Specification Workflow

The component developer's workflow is only sketched here, since it is performed separately from the software architect's workflows. If a new component needs to be implemented, the workflow of the component developer (see Figure 2.5) can be assumed to be part of the provisioning workflow according to Cheesman and Daniels [10].

Any development process model can be used to construct new components as long as functional and extra-functional properties are specified properly. First, a *component requirements analysis* has to be conducted. It is succeeded by *functional property specification* and then *extra-functional property specification*. The functional properties consist of interface specifications (i.e., signatures, pre/postconditions, protocols), descriptions of internal dependencies between provided and required interfaces. We use service effect specifications (Section 3.1.4) to describe such dependencies. Additionally, descriptions of the functionality of component services have to be made. Extra-functional, QoS-relevant information includes resource demands, reliability values, data flow, and transition probabilities for service effect specifications. Finally, after *component implementation* according to the specifications, component developers have to put the binary implementations and the specifications into repositories, where they can be retrieved and assessed by software architects.

The specification workflow of the software architect consists of four inner workflows. The first two workflows (*component identification* and *component interaction*) are adapted from [10] except that we explicitly model the influence on these workflows by existing components. For component identification, so-called `ProvidedComponentTypes` can be used in the PCM (cf. Section 3.1.3.3). Component interaction can be described in the PCM once the provided component types have evolved to `ImplementationComponentTypes` (cf. Section 3.1.3.1). During the *component specification*, the software architect additionally gets existing interface and service effect specifications as input. Both are transferred to the new workflow *interoperability check*. In this workflow, interoperability problems are solved and the architecture is optimised. For example, functional parametrised contracts [12], which are modelled as service effect specifications, can be computed (cf. Section 2.3). The outputs of the specification workflow are an architecture and component specifications with refined interfaces.

2.1.5 QoS Analysis Workflow

During QoS analysis, the software architecture is refined with information on the deployment context, the usage model, and the internal structure of components. Figure 2.6 shows the process in detail.

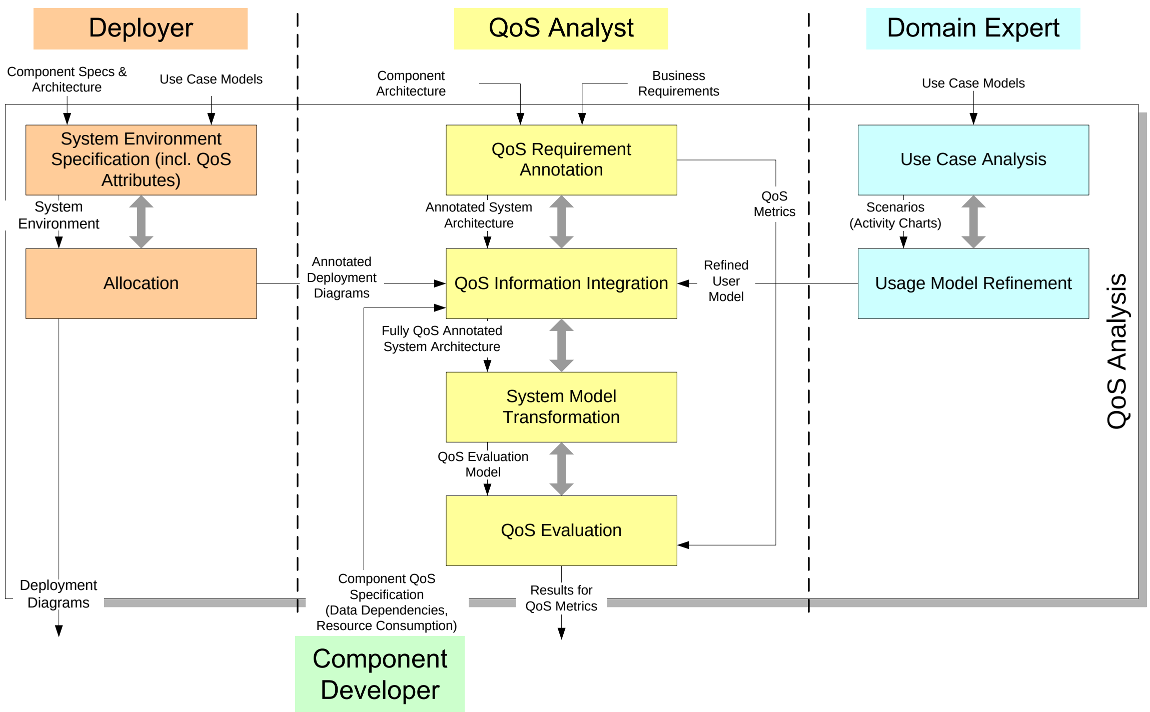


Figure 2.6: Detailed View of the QoS Analysis Workflow

The deployer starts with the *resource environment specification* based on the software architecture and use case models. Given this information, the required hardware and software resources and their interconnections are derived. As a result, this workflow yields a description of the resource environment, for example, a deployment diagram without allocated components or an instance of the resource environment model (cf. Section 3.3.4). Instead of specifying a new resource environment, the deployer can also use the descriptions of existing hardware and software resources. Moreover, a set of representative system environments can be designed if the final resource environment is still unknown. For QoS analysis, detailed information on the resources modelled in the environment are required.

During *allocation*, the deployer specifies the mapping of components to resources. The result of this workflow can be a complete deployment diagram or a resource environment plus allocation contexts for components as described in section 3.3.5. The resulting specifications are part of the inputs of the QoS analysis models used later. The resulting fully specified resource environment and component allocation are passed to the QoS analyst.

The domain expert refines the use case models based on the requirements during the *use case analysis* workflow. A description of the scenarios for the users is created based on an external view of the current software architecture. The scenarios describe how users interact with the system and which dependencies exists in the process. For example, activity charts or usage models (cf. Section 3.4.2) can be used to describe such scenarios. The scenario descriptions are input to the *usage model refinement*. The domain expert annotates the descriptions with, for example, branching probabilities, expected size of different user groups, expected workload, user think times, and parameter characterisations.

As the central role in QoS analysis, QoS analysts integrate relevant information, perform evaluations, and deliver feedback to all involved parties. In the *QoS requirement annotation* workflow, the QoS analyst maps QoS requirements to direct requirements of the software architecture. For example, the maximum waiting time of a user becomes the upper limit of the response time of a component's service. While doing so, the QoS analyst selects metrics, like response time or probability of failure on demand, that are evaluated during later workflows.

During *QoS information integration*, the QoS analyst collects the specifications provided by the component developers, deployers, domain experts, and software architects, checks them for soundness, and integrates them into an overall QoS model of the system. In case of missing specifications, the QoS analyst is responsible for deriving the missing information by contacting the respective roles or by estimation and measurement. The system specification is then automatically transformed into a prediction model.

The *QoS evaluation* workflow either yields an analytical or simulation result. QoS evaluation aims, for example, at testing the scalability of the architecture and at identifying bottlenecks. The QoS analyst performs an interpretation of the results and comes up with possible design alternatives. Automated search approaches can help the QoS analyst to improve the architecture, e.g. by optimising the deployment. Finally, the QoS analyst delivers the results to the software architect. If the results show that the QoS requirements cannot be fulfilled with the current architecture, the software architect has to modify the specifications or renegotiate the QoS requirements.

2.1.6 Discussion

Related Work There are numerous publications on component-based development processes [6, 13, 14, 15, 16, 2]. However, most of these process descriptions do not deal with extra-functional properties. Furthermore, there are many approaches for the performance prediction of component-based software systems [11], but only few describe the encompassing development process in detail or spread the needed information for QoS analyses among the participating roles.

Role Names There are several synonyms for the role names we have chosen for the PCM process model.

- **Component Developer**: Application Component Provider, Component Implementer, Component Programmer
  We chose ”component developer” because it is quite generic, should be known to most software engineers, and describes the tasks of this role best.
- **Software Architect**: System Architect, Component Assembler, System Assembler, Architect, Application Assembler
  It might be argued that this role does not only deal with software, but also has an influence on the hardware environment. Therefore the broader term ’system’ instead of ’software’ could be used. However, the term software architect is quite established and the tasks involving the hardware environment of the component-based system could be delegated to the deployer. The term component assembler is sometimes used in the literature, it is, however, a too restricted term for the tasks of this role.
- **Deployer**: System Allocator, Component Deployer, Assembly Allocator, System Administrator, Resource Specifier, Execution Environment Modeller, Middleware Expert, Deployment Expert
  In J2EE the term ’deployment’ is used for assembling components and allocating them on resources. In the PCM, we explicitly separate between assembly and allocation, as the former is conducted by the software architect and the latter by the deployer. We chose the most generic term ’deployer’ for the role, which is responsible for specifying the resource environment and allocating component assemblies to resources.
- **QoS Analyst**: Performance Analyst, Reliability Specialist, QoS Expert, QoS Evaluator, QoS Manager
  As we do not want to restrict our model to performance or reliability analyses we chose the collective term ’QoS’ (Quality-of-Service), which covers performance, reliability, availability, etc. The goal of this role is to come up with analyses of the QoS properties of an application, so we chose the term ’QoS analyst’
- **Domain Expert**: Business Expert, Usage Modeller
  We are not sure, if there are dedicated roles for specifying user behaviour in IT organisations. Therefore we chose the term ’domain expert’, because this role might be involved into other tasks related to requirements analyses in addition to the task of usage modelling.

Is there a QoS Analyst? As described in Section 2.1.2, the role of QoS analysts bundles the tasks of 1) deriving QoS information from the requirements, 2) integrating information from the other roles, 3) estimating missing input parameters, 4) using QoS analysis tools such as queueing network solver, and 5) pre-interpreting the results of these tools.

It can be argued that this role is not really necessary, as most of the tasks could also be performed by the software architect. In fact, task 2), 4), and 5), should even be encapsulated into user-friendly tools, so that no special knowledge would be required to perform the QoS analysis. Task 3) might require special knowledge of a QoS domain, but software architects should at least be able to provide rough estimation for missing values, which might be sufficient for early QoS analysis.

However, it can also be argued, that existing tools are not so far advanced to automate the tasks of this role. The manual specification of additional input parameters for the prediction method might be too time-consuming and thus expensive for software architects. Additionally, it remains questionable if task 5) can be encapsulated into tools as it is sometimes difficult to map analysis or simulation results to problems in the architecture. Furthermore, designing QoS-improving architectural alternatives requires special knowledge (such as performance patterns or configuration options of component containers).

We have decided to keep the role in the model, because of role-based separation of concerns. We suppose that today the QoS analysis task is often delegated to specialists.

2.2 Interfaces and Composition

2.2.1 Interfaces as First-Class Entities

According to Parnas [17], an interface is an abstraction of piece of software (a software entity) which should contain a sufficient amount of information for a caller to understand and finally request the realised functionality from any entity claiming to offer the specified functionality. Note that this implies, that the specification of the interface also has to contain a sufficient amount of information for the implementer to actually implement the interface. Due to the inherent need of an interface to abstract the behaviour of the software entity not in all cases there is sufficient information provided to use or implement an interface in an unambiquious way.

This definition has several consequences. First of all, interfaces can exist on their own, i.e., without any entity requesting or implementing the specified functionality. In industrial practice, this is actually often used. For example, Sun defined for the Java programming language several sets of Java interfaces which deal with a specific sets of generic functionality without actually providing an implementation. Part of these domain-standards are the Java Messaging Standard dealing with different types of message-based communication or the Java Persistence API concerned with persisting objects in relational databases.

Second, two roles can be identified a software entity can take relative to an interface. Any entity can either claim to implement the functionality specified in an interface or to request that functionality. This is reflected in the Palladio Component Model by a set of abstract meta-classes giving a conceptual view on interfaces, entities and their relationships. The abstract meta-class `InterfaceProvidingEntity` is inherited by all entities which can potentially offer interface implementations (Figure 2.7). Similarly, the meta-class `InterfaceRequieringEntity` is inherited by all entities which are allowed to request functionality offer by entities providing this functionality. Details follow in Section 3.1.

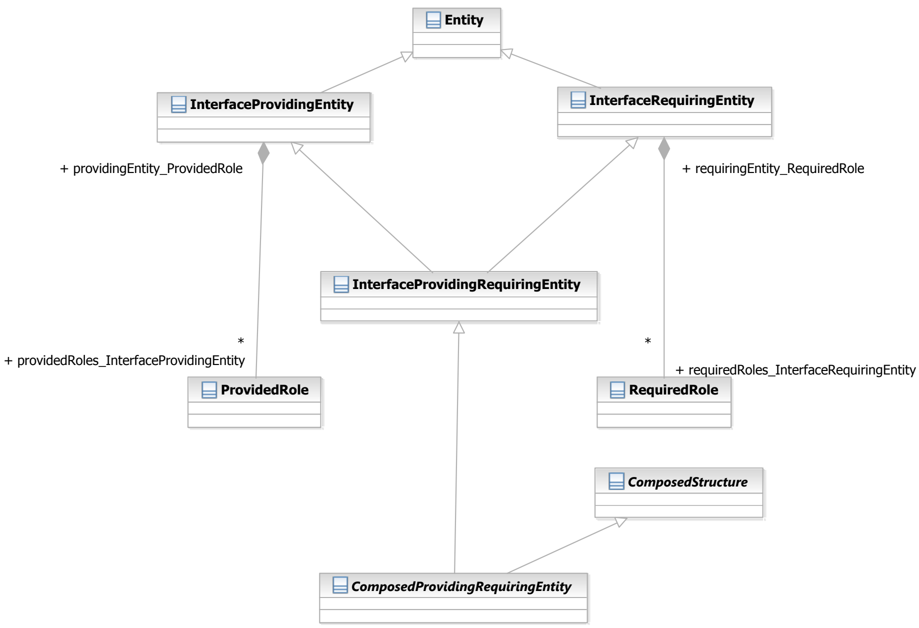


Figure 2.7: `InterfaceProvidingRequiringEntity` in the PCM metamodel

2.2.2 Composed Structure

Clements Szyperski approaches the definition of a component in his book on component based software-engineering with the statement "Components are for composition, much beyond is unclear" [6]. This statement does not only point out how hard it is to find a common definition for the term "component". It also highlights the most common principle in the definitions of the term component in the literature: a component can be composed with other components in order to get a more complex structure. We also consider the ability to compose components into new structures as a primary feature of components. As a consequence, our component model contains an abstract conceptual view of the concept of a `Composed-Structure`, which is a structure build by composing components (Figure 2.8, details in Section 3.2).

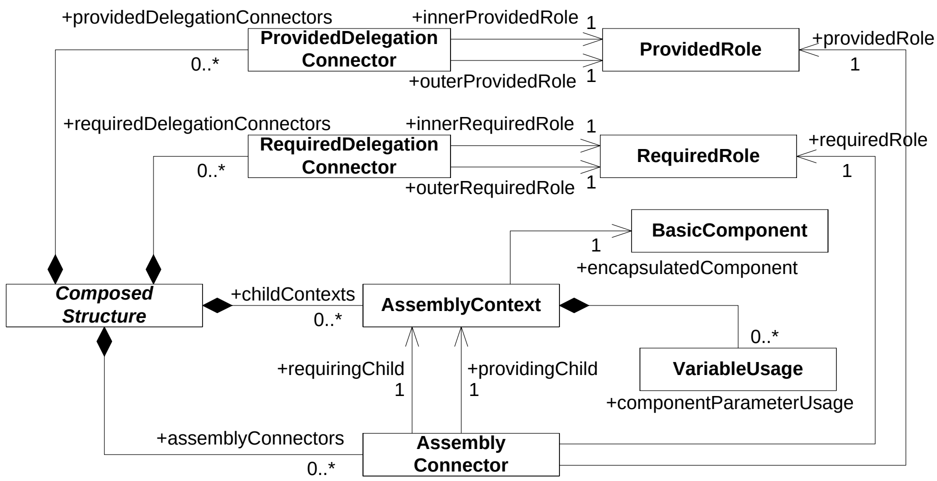


Figure 2.8: `ComposedStructure` in the PCM metamodel

2.3 Parametric Contracts

2.3.1 Classical Contracts for Software Components

Before defining contracts for components, we briefly review B. Meyer's design-by-contract principle from an abstract point of view. According to [18, p. 342], a contract between the client and the supplier consists of two obligations:

- The client has to satisfy the precondition of the supplier.
- The supplier has to fulfill its postcondition, if the precondition was met by the client.

Each of the above obligations can be seen as the benefit for the other party. (The client can count on the postcondition if the precondition was fulfilled, while the supplier can count on the precondition). Putting it in one sentence:

> If the client fulfills the precondition of the supplier, the supplier will fulfil its postcondition.

The used component plays the role of a supplier. But to formulate contracts for components, we also have to identify the pre- and postconditions and the user of a component. But what is to be considered a precondition, postcondition and user depends on whether the component is used at run-time or configuration-time. Let's first consider the component's use at run-time. Using a component at run-time is calling its services. Hence, the user of a component C is the set of all components connected to C's provides-interface(s).

The precondition for that kind of use is the precondition of the service, likewise the postcondition is the postcondition of the service. Actually, that shows that this kind of use of a component is nothing different as using a method. Therefore, the author considers this case as the use of a *component service*, but *not* as the use of a *component*. Likewise, the contract to be fulfilled here from client and supplier is a *method contract* as described by B. Meyer already in 1992. This is the contract for using a *component service*, but not the contract for using the *component*!

The other case of component usage (usage at composition-time) is actually the relevant case when talking about the contractual use of components. This is the important case when architecting systems out of components or deploying components within existing systems for reconfigurations. Again, in this case a component C is acting as a supplier and the component connected to the provides interface(s) as a client. The component C offers services to the those components of the assembly context which are connected to C's provides-interface(s). According to the above discussion of contracts, these offered services are the postcondition of the component, i.e., what the client can expect from a working component. Also according to B. Meyer's above mentioned description of contracts, the precondition is what the component C expects from those components of the assembly context which are connected to C's requires-interface(s) to be provided by the assembly context, in order to enable C to offer its services (as stated in its postcondition). Hence, the precondition of a component is stated in its requires-interfaces. Analogously to the above single sentence formulation of a contract, we can state:

> If the user of a component fulfills the component's requires-interface (offers the right required components in the assembly context) the component will offer its services as described in the provides-interface.

Let us denote with pre_c the precondition of a component c and with $post_c$ the postcondition of a component c. For checking whether a component c can be replaced safely by a component c', one has to ensure that the contract of c' is a subcontract of c. The notion of a subcontract is described in [18, p. 573] like contravariant typing for methods: A contract c' is a subcontract of contract c iff

$$pre_{c'} \trianglelefteq pre_c \wedge post_{c'} \trianglerighteq post_c \tag{2.1}$$

(Where $\trianglerighteq$ means "stronger", i.e., if pre_c and $post_c$ are predicates,$\trianglerighteq$ is the logical implication $\Rightarrow$. In the set semantics of pre- and postcondition below, $\trianglerighteq$ is the inclusion $\supseteq$.)

To check the interoperability between components c and c' (see point (1) in figure 2.9), one has to check whether

$$pre_c \trianglelefteq post_{c'} \tag{2.2}$$

Coming back to protocol-modelling interfaces, we can consider the precondition of a component as the set of required method call sequences, while the postcondition is the set of offered call sequences. In this case, the checks described in the above formulas (2.1) and (2.2) boiled down to checking the inclusion relationship between the sets of call sequences, i.e., for the substitutability check we have:

$$pre_{c'} \subseteq pre_c \wedge post_{c'} \supseteq post_c \tag{2.3}$$

and for the interoperability check:

$$pre_c \subseteq post_{c'} \tag{2.4}$$

For arbitrary sets A and B holds $A \subseteq B \iff A \cap B = A$. Hence, the inclusion check we need for checking interoperability and substitutability can be reduced to computing the intersection and equivalence of sets of call sequences. One of the main reasons for choosing finite state machines (FSMs) as a model to specify these sets of call sequences was the existence of efficient algorithms for computing the intersection of two FSMs and checking their equivalence. Of course, more powerful models than FSMs exist (in the sense that they can describe protocols which cannot be described by FSMs) but for many of these models (like the various push-down automata) the equivalence is not decidable (see e.g., [19]). Hence, one can use these models for specifying component interfaces, but that does not help to check their interoperability or substitutability at configuration-time.

2.3.2 Parametric contracts as a generalisation of classical contracts

While interoperability tests check the requires-interface of a component against the provides-interface of *another* component, parametric contracts link the provides-interface of one component to the requires-interface of *the same* component (see figure 2.9).

The usefulness of parametric contracts is based on the observation that in practice often only a subset of a component's functionality is used. This is especially true for coarse-grained components. In this case, also only a subset of the functionality described in the requires-interface is actually used. That means that the component could be used without any problems in assembly contexts where not all dependencies, as described in the requires interface, are fulfilled. Vice versa, if a component does not receive all (but some) functionality it requires from the assembly context, it often can deliver a reasonable subset of its functionality.

These facts can be modelled by a set of possible provides-interfaces $\mathbf{P} := \{prov\}$ and a set of possible requires-interfaces $\mathbf{R} := \{req\}$ and a monotone total bijective mapping p between them $p : \mathbf{P} \to \mathbf{R}$.[1] As a result, each requires-interface $req \in \mathbf{R}$ is now a function of a provides-interface $prov$: $req = p(prov)$

[1] p can be made total and surjective by defining $\mathbf{P} := dom(p)$ and $\mathbf{R} := im(p)$.

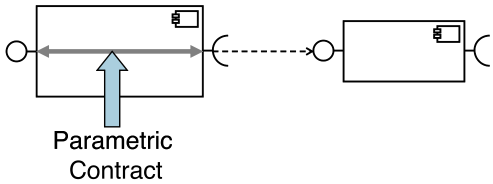


Figure 2.9: Parametric Contracts

and (because p is bijective) each provides-interface $prov \in \mathbf{P}$ can be modelled as a function of a requires-interface $req \in \mathbf{R}$: $prov = p^{-1}(req)$.

This mapping p is now called *parametric contract*, since it parameterises the precondition with the postcondition of the component and vice versa. It can be considered as a generalisation of "classical contract" which uses a fixed pre- and postcondition. The parametric contract is bundled with the component and computes the interfaces of the components on demand.

For the following, assume component B uses component C and is used by component A. If component A uses only a subset of the functionality offered by B we compute a new requires-interface of B with the parametric contract p_B:

$$p_B(req_A \cap prov_B) =: req'_B \subseteq req_B \tag{2.5}$$

Note that the new requires-interface req'_B requires possibly less than the original requires-interface $req_B := p_B(prov_B)$ (but never more) since p_B is monotone and $req_A \cap prov_B \subseteq prov_B$. When computing the requires-interface out of a provides-interface (possibly intersected with an external requires-interface) the parametric contract is called *provides-parametric contract*.

Likewise, if component C does not provide all the functionality required by B, one can compute a new provides-interface $prov'_B$ with p_B:

$$p_B^{-1}(req_B \cap prov_C) =: prov'_B \subseteq prov_B \tag{2.6}$$

Since p_B is monotone, p^{-1} is, too. With $req_B \cap prov_C \subseteq req_B$ we have $prov'_B \subseteq prov_B := p^{-1}(req_B)$. In this case we use a *requires-parametric contract*.

Technically, the parametric contract is specified by the service effect specification. The actual way what to specify to calculate the parametric contract depends on the interface model used. In case of protocol modelling interfaces, the service effect specification can be given by FSMs [20]. In case of quality of service modelling interfaces, only requires parametric contracts are used. This is because a provides parametric contract evaluates not to a concrete interface with QoS requirements, but to constraints which describe a set of possible requires interfaces. Anyhow, for QoS modelling interfaces the parametric contract is given by service effect specifications, as described in section 3.1.4.

2.4 Context

2.4.1 Motivation

One of the most important arguments for component-based software development is the black-box reuse of components. Components are developed by third party vendors, who sell their products to multiple clients. Therefore, component developers cannot make assumptions on the underlying operating system and hardware as well as the usage profile and connected components. In other words, the context the component will be used in is unknown to component developers. Szyperski defines a software component as "a unit of composition with contractually specified interfaces and explicit context dependencies only. A software component can be deployed independently and is subject to a composition by third parties" [6].

This definition emphasises the importance of context dependencies and their explicit definition. However, it remains vague what is actually part of the context beyond the relationships defined by the provided and required interfaces of a component. One of those undefined dependencies is the underlying hardware that influences QoS attributes of a component, like performance and reliability. Especially for QoS predictions, knowledge about such dependencies to the context is needed in addition to functional specifications, like behavioural protocols [21] and service effect specifications [22].

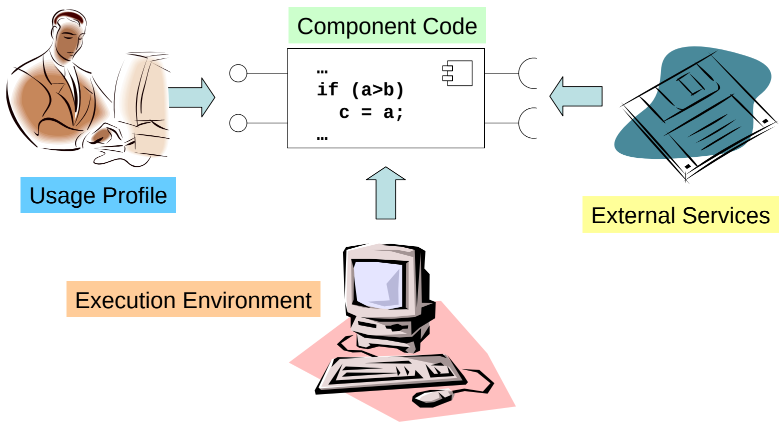


Figure 2.10: Influences on QoS properties of a software component.

Factors influencing the QoS attributes of a component can be classified into four main categories as shown in figure 2.10:

1. The implementation of the component, e.g. the selection of an algorithm.
2. The quality of required services, e.g. calling a slow or a fast service will result in a different performance for the provided service perceived by a user.
3. The runtime environment the component is deployed in. This includes the hardware and system software like the operating system and middleware platforms.
4. The usage of the component, e.g. if the component has to serve many requests per time span it is more likely to slow down.

With these four categories of influences we can define the quality of a provided service *s* of a concrete component as a function of the varying influences. The implementation of the component's service is considered as a constant as it does not depend on its context but is fixed by the component developer at implementation time. Thus, the QoS of a component can be defined as a function of the remaining parameters, which are determined during its allocation, assembly and usage:

$$q_{impl} : \mathscr{P}(s) \times DR \times UP \rightarrow Q$$

where $\mathscr{P}(s)$ is the domain of the set of external services used by service s, DR specifies the deployment relationship defining which component and connector is deployed on which part of the execution environment and UP describes the usage profile. As a result, the function yields a value in the domain of the investigated quality metric Q.

2.4.2 Context Influences

Since QoS attributes of a component are strongly influenced by the environment the component is used in, the actual delivered QoS can only be determined knowing all influencing factors. We identified three aspects defined during system design that determine the complete context of a component based on the influences shown in figure 2.10: composition (connected components), usage, and allocation. For understandability, we split the influence of composition into the parts hierarchy and system/assembly and leave out the influence of the usage profile. All aspects are associated to different roles in the component-based development process as described in section 2.1.

The structure of a system/assembly is defined by software architects who decide which components are used and how they are connected. Similarly component developers may construct composite components, which define the hierarchy of the system. Deployers define the execution environment and allocate software components among different resources, like servers and desktop computers.

System/Assembly (Horizontal Composition) A system specifies which components are used within an application and how they communicate. Within the system, the required interfaces of components are connected to provided interfaces of other components. That way it is determined which concrete external services are called by a component.

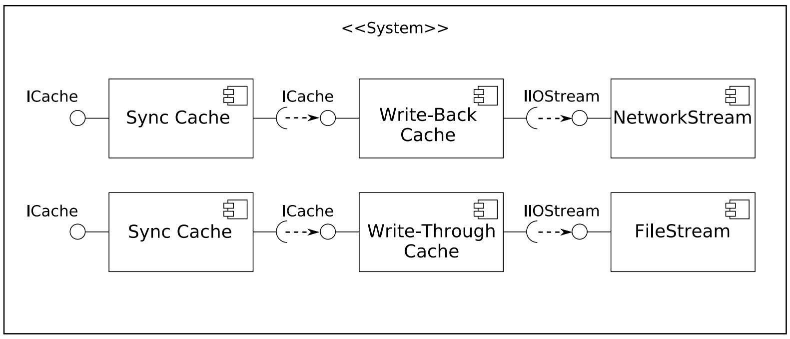


Figure 2.11: Component assembly.

A component can be used multiple times within a single system. Figure 2.11 illustrates this with a simple example. Three different types of components exist in the system shown there. On the right hand side, we have two I/O components that manage the access either to a file or network connection.

Two different kinds of caching components that implement different caching strategies are shown in the middle. The `SyncCache` component on the left-hand side allows multiple tasks to access the caches concurrently without producing an incorrect state of the connected single-threaded caches.

The same component (`SyncCache`) is inserted at two different locations within the system. Both representations of the component are connected differently. Thus, users or other components that call the services provided by the different component representations will experience different QoS on the provided interfaces of the respective component representations. This is caused by the different caching strategies and I/O devices used by the `SyncCache` components. Modelling the component context explicitly allows us to hold the information on the diverse connections and the resulting quality attributes without changing the component specification.

Hierarchy (Vertical Composition) Related to the system, another important part of the context is the hierarchy in which a component is used. In figure 2.12, a composite component (`BillingManager`) is depicted which has been designed to create bills and store each one in a single PDF (Portable Document Format) file. The component is additionally supposed to write a summary of all the created bills as PDF file. Hence, the component `PDFCreator` is used in two different places. Notice however, that this kind of usage is usually unknown to the creators of the outer composite component. For them, the inner component (`BillCreator`) is a black box. They do not know the internal details and, hence, the usage of the inner `PDFCreator` is hidden.

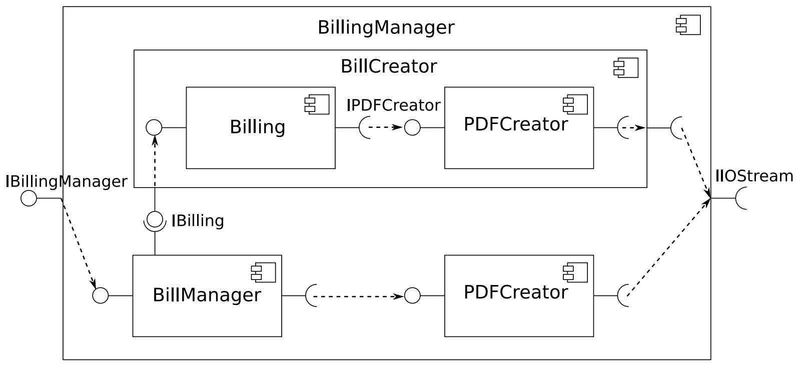


Figure 2.12: Component hierarchy.

In this case, the `PDFCreator` component is used in different contexts on different hierarchy levels. Note, that this only makes sense if the underlying component model supports hierarchical components at all. Considering parametric contracts, both components might offer different characteristics (QoS, functions offered, etc.). Additionally, they are *used* differently in their contexts. The `PDFCreator` of the inner component produces bills with less pages than the summary PDF file created by the outer `PDFCreator`.

Allocation An explicit context model is especially advantageous to model the allocation of components on hardware and software resources. Figure 2.13 depicts a system that uses replicated components to fulfil requests. In our example, server I is assumed to be slow and server II is assumed to be fast. Hence, the workload is not distributed equally, but 30% of the requests are directed to server I and 70% are directed to server II.

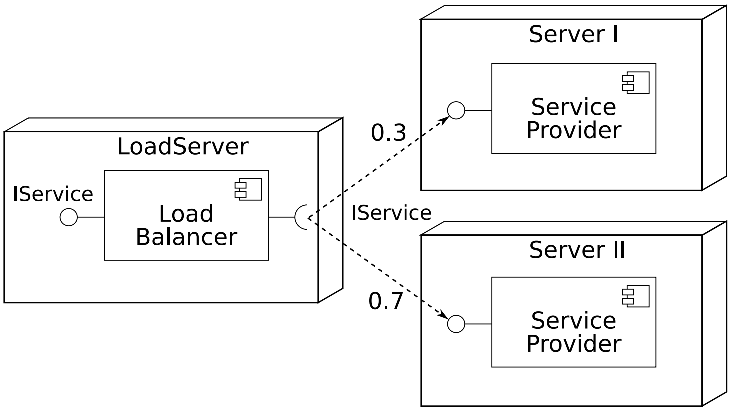


Figure 2.13: Component allocation.

Here, we see several context influences. We have two copies of the same component allocated on different machines and, thus, in different contexts. The workload of each replicated component is different because of the distribution strategy. The processing power available to both replicated components is varying with the underlying hardware systems. However, both components are connected with an identical logical link going from the required interface of the workload balancer to the provided service of the replicated component. But again, each of these logical connections is most likely using a different physical communication channel, i.e., different network links.

2.4.3 Context Model

The following section is taken from [23].

The PCM features a so-called context model, which includes information about a component's binding, allocation, and usage. While component developers supply individual component specification, further information about each component in an architecture is necessary for QoS predictions. This information is only available after component implementation and cannot be supplied by the component developer. Therefore, the PCM allows separate creation of the context model by other developer roles or tools.

Before describing the context-model's implementation in the PCM, first the general concept shall be explained. The context-model includes manually specifiable and computable parts (Tab. 2.1).

The assembly context refers to a component's binding to other components. The manually specifiable part includes both the connections to other components via provided and required interfaces and the containment relationship between vertically composed components. The computable part refers to the parametric contracts introduced by Reussner [12]. For example, they allow to restrict the set of required services of a component if certain provided services are not needed. A `BasicComponent` can have multiple assembly contexts in the same architecture. In this case, each assembly context refers to a copy of the same component implementation.

The allocation context refers to a component's binding to hardware/software resources. It requires manual specifications of a component's allocation and configuration options related to hardware and software resources. Tools can compute allocation-dependent QoS characteristics of a component by combining information from the component specification and the hardware environment. For example, a

	Assembly Context	Allocation Context	Usage Context

Specified	• Horizontal Composition: Binding to other Components
	• Vertical Composition: Encapsulation in Composite Components
		• Allocation to Hardware Resources
		• Configuration
		  ○ Component
		  ○ Container
		  ○ Concurrency
		  ○ Communication
		  ○ Security
		  ○ …
			• Usage at System Boundaries
			  ○ User Arrival Rate
			  ○ Number of Users
			  ○ Request Probabilities
			  ○ Parameter Values

Computed	• Parametric Contracts
	  ○ Provided/Required Services
	  ○ Provided/Required Protocols
	  ○ …
		• Allocation-dependent QoS Characteristics
		  ○ Timing Values for Resource Demands
		  ○ …
			• Usage inside Components
			  ○ Branch Probabilities
			  ○ Loop Iteration Numbers
			  ○ Input/Output Parameters

Table 2.1: Component Context relevant for QoS Prediction

resource demand provided by a component developer can be transformed into a timing value, if the speed of the underlying hardware resource is known. A `BasicComponent` can have multiple allocation context in the same architecture, in which case each allocation context refers to a copy of the same component implementation running on different resources.

The usage context refers to a component instance's usage by clients. For PCM instances, only the user behaviour at the system boundaries needs to be specified. This includes the number of users, which services they call, and what parameter values they supply. Tools can then propagate these values to each component specification in the architecture and compute individual component usage including branch probabilities, loop iteration numbers, and parameter values.

2.5 Random Variables

2.5.1 Overview

In the Palladio Component Model, we use *random variables* in expressions which specify parametric dependencies. The rationale behind this is that many aspects of larger software systems, especially in the business information systems area, can not be modelled having complete information (see for a classification of the information types for example [24]). Uncertainties can be found in many aspects of the system model. Two main sources stem from the behaviour of users and time spans of method executions (because we do not consider real-time environments). User behaviour can only be specified in a stochastical manner. How long users think (aka think time) between requests to the system, what parameter values they use in their requests can often only be characterized using probabilities. The second source comes from the facts that the PCM is designed to support predictions on an architectural level. On such a level, real-time constraints are usually not available. The reasons are also manifold. First, the information is simply not available at early design stages. Second, environmental features as garbage collectors, middle-ware services, etc. make it hard to predict timing time consumptions with certainty.

In the following, the use of random variables in parametric dependencies is introduced. These types of specifications are used in several places in the PCM. They are used especially in the ResourceDemandingSEFF and the ResourcePackage to describe resource consumptions. A library in the implementation of the PCM supports the use of random variables in different types. See 2.5.5 for technical details on this library.

2.5.2 Definition

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

$$X : \Omega \rightarrow \mathbb{R}$$

with Ω = 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. Note, that often a random variable has a certain unit (like seconds or number of bytes, etc.). It is important for the user of prediction methods to keep the units in the calculations and in the output to increase the understandability of the results.

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. A probability distribution yields the probability of X taking a certain value. It is often abbreviated by $P(X = t)$. For discrete random variables, it can be specified by a probability mass function (PMF). For continuous variables, a probability density function (PDF) is needed. However, for non-standard PDFs it is hard to find a closed form (a formula describing the PDF). Because of this and for reasons of computational complexity, we use discretisized PDFs in our model.

For the event spaces Ω we support include integer values $\mathbb{N}$, real values $\mathbb{R}$, boolean values and enumeration types (like "sorted" and "unsorted") for discrete variables and $\mathbb{R}$ for continuous variables.

2.5.3 PDF discretisation

A probability density function (PDF) represents a probability distribution in terms of integrals. The probability of an interval $[a,b]$ for a pdf $f(x)$ is given by the integral

$$\int_a^b f(x)dx$$

for any two number a and b, $a < b$. To fulfill this property, $f(x)$ has to be a non-negative Lebesque-integrable function $\mathbb{R} \to \mathbb{R}$. The total integral of $f(x)$ has to be 1.

To create a discrete representation of a PDF, we use the fact that the probability of an interval is given by its integral. Basically, there are two ways to approximate a PDF, which mainly differ in the way how the intervals are determined. The first one uses a sampling rate and, therefore, a fixed interval size. The second one uses arbitrary sizes for intervals. Both methods store the probabilities for the intervals, not the probability density.

2.5.3.1 Sampling – Fixed intervals

To create an approximation of a PDF by a set of fixed intervals, the domain of the PDF is devided into N intervals denoted by the set I, each of which has the same width specified by the value d. The ith interval is then defined by $[(i-1/2)d,(i+1/2)d[$. For our purposes, we can assume that the domain of a PDF is always greater or equal to zero. Thus, we set the first interval ($i=0$) to $[0,1/2d[$. To minimize computational errors, we associate the probability of the ith interval $[(i-1/2)d,(i+1/2)d[$ to its middle value $i*d$. So, we get a set of N probabilities, where the probability of interval i, p_i is given by the integral:

$$p_i = \int_{(i-1/2)d}^{b} f(x)dx, \qquad lim_{b \to (i+1/2)d}$$

for $i > 0$ and

$$p_i = \int_0^b f(x)dx, \qquad lim_{b \to 1/2d}$$

for $i = 0$. The approximation of a PDF is completely described by the interval width d and the probabilities p_i for all intervals I.

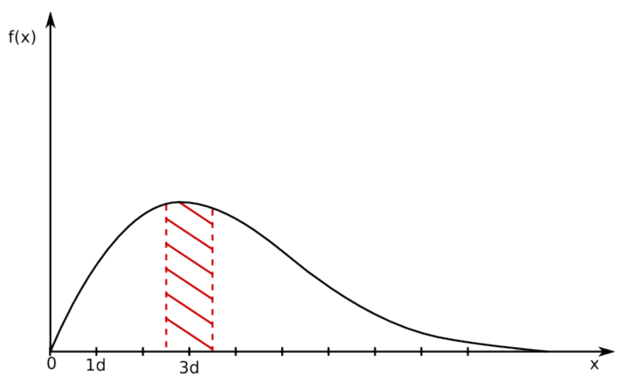


Figure 2.14: The probability of the interval $[2.5d,3.5d[$ is the striped area under the graph.

Example 2.1. Figure 2.14 illustrates how a pdf $f(x)$ is approximated by deviding its domain into a set of intervals. The X-axis shows the multiples of the interval width d ($1d$, $2d$, $3d$...). These values represent the mean values of the intervals to which the probabilities will be associated. The figure shows how the probability of the third interval is computed. The interval borders are given by $[2.5d, 3.5d[$ and its mean value is $3d$. The probability p_3 is the striped area under the graph. So, all values lying in this area are associated to the value $3d$.

2.5.3.2 Approximation by boxes – Variable intervals

For many PDFs, variable interval sizes allow a better approximation using less values compared to fixed ones. This is especially useful if the function consists of large, almost constant parts and sharp peaks on the other hand. Variable interval sizes allow the specification of almost constant areas by one large interval and the use of multiple, fine grained intervals for sharp peaks, which need to be described in more detail.

We have a set of intervals I so that for each two intervals $J_1, J_2 \in I$, $J_1 \neq J_2$ the disjunction is the empty set $J_1 \cap J_2 = \emptyset$ and the union of all intervals forms a new interval from zero to $x \in \mathbb{R}^+$, $\cup_{J \in I} = [0, x[$. Intuitively, this means that the intervals do not overlap and that there are no gaps between the intervals.

To ensure both properties mentioned above, the intervals are specified by their right hand value only. Thus, we have a set I_X whose values define the right hand sides of all intervalls. Suppose we can define an order on the set such that $x_1 < x_2 < \ldots < x_{n-1} < x_n$. Then the ith interval is $[x_{i-1}, x_i[$ for $i > 1$ and $[0, x_1[$ for $i = 1$. This allows us to specify n intervals by n values only and to ensure that the intervals neither do overlap nor have gaps inbetween. Now the probability p_i for the ith interval is given by

$$p_i = \int_{x_{i-1}}^{b} f(x)dx, \qquad lim_{b \to x_i}$$

for $i > 1$ and

$$p_i = \int_{0}^{b} f(x)dx, \qquad lim_{b \to x_1}$$

for $i = 1$.

2.5.4 Functional random variables

2.5.4.1 General

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 mathematical operations ($*, -, +, /, ...$) as well as some logical operations for boolean type expressions ($==, >, <,$ `and`, `or`, ...).

To give an example, the distribution of a random variable N is depicted in figure 2.15. The variable could model some characterisation of the size of a parameter of a component service.

To determine the time consumption of the method body which depends on the characterisation N it is known that the amount of CPU instructions needed to execute the method is three times N. The resulting distribution function is shown in figure 2.16.

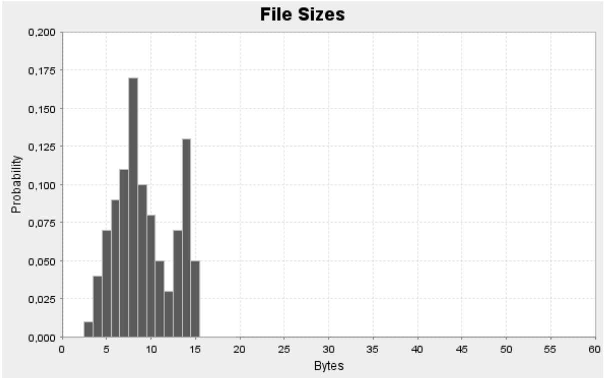


Figure 2.15: A Distribution of a Discrete Random Variable N

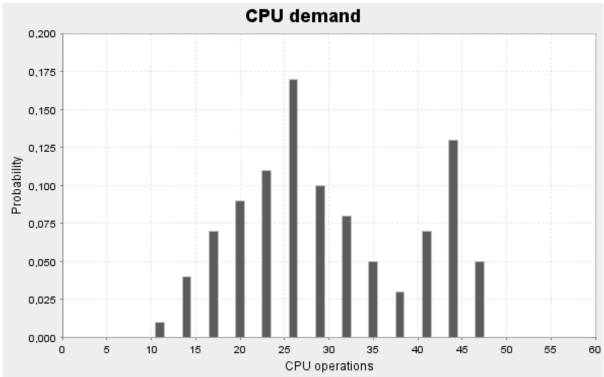


Figure 2.16: A Distribution of N * 3

2.5.4.2 Differences btw. discrete and continuous variables

As introduced above we support the use of discrete as well as continuous variables. However, in such a case special care has to be taken when constructing expressions. Three times of a discrete variable can not be determined in the same way as three times a continuous variable. The reason for this is that continuous variables are also scaled continuously. To give an example, consider the continuous variable X which is uniformly distributed in a range between 5 and 10 seconds. If the variable is now multiplied by three, possible values of the resulting random variable can be in the interval between 15 and 30 seconds. The resulting distribution is again uniformly distributed having a density function which is one-third of the original density function.

An analogous example for a discrete random variable follows. Consider a discrete random variable taking the value 5 in 30% of all cases, 7 in 20% of all cases and 10 in the remaining 50%. If this variable is multiplied by 3, the result is a variable taking the value 15 in 30%, 21 in 20% and 30 in 50% of all cases. The probabilities of the single events stay the same only the actual outcome changes.

The depicted difference is especially important in the case of discretisized PDFs. Any mathematical operation in which such a variable is involved has to treat the discretisized PDF as a 'real' PDF in order to avoid calculation mistakes.

2.5.5 Stochastic Expressions

We call the language in which functional random variables can be specified Stochastic Expressions. As said before, the specifications of this kind of expressions is based on mathematical operations like addition or multiplication. The complete grammar is given below.

The StoEx-framework includes a meta-model with a textual, concrete syntax to combine multiple random variables with different operations (e.g., addition, multiplication) to define new random variables. 2.17 depicts the meta-model and thus illustrates the abstract syntax of the so-called "Stochastic Expression Language" implemented by the StoEx-framework. RandomVariables contain their specification both as a string (attribute specification) and as a derived attribute, which yields an instance of the Expression meta-class possibly being a large object tree realising the abstract syntax tree of the stochastic expression. Developers enter the string representation into tools, from which the lexer and parser of the StoEx-framework create the object representation used for computations or model-transformations.

2.5.5.1 Parser (EBNF)

```
expression : compareExpr;
compareExpr : sumExpr( ( GREATER | LESS | EQUAL | NOTEQUAL | GREATEREQUAL | LESSEQUAL) sumExpr | );
sumExpr : prodExpr ( ( PLUS | MINUS ) prodExpr )*;
prodExpr : powExpr ( ( MUL | DIV | MOD ) powExpr )*;
powExpr : atom( POW atom|);
atom: ( NUMBER | scoped_id | definition | STRING_LITERAL | boolean_keywords | LPAREN compareExpr RPAREN);
scoped_id : ID ( DOT ( ID | "INNER") )*;

definition : "IntPMF" LPAREN ( unit ) RPAREN SQUARE_PAREN_L ( numeric_int_sample )+ SQUARE_PAREN_R
| "DoublePMF" LPAREN ( unit ) RPAREN SQUARE_PAREN_L ( numeric_real_sample )+ SQUARE_PAREN_R
| "EnumPMF" LPAREN ( unit )( SEMI ORDERED_DEF|)RPAREN SQUARE_PAREN_L ( stringsample )+ SQUARE_PAREN_R
| "DoublePDF" LPAREN ( unit ) RPAREN SQUARE_PAREN_L( real_pdf_sample )+ SQUARE_PAREN_R
| "BoolPMF" LPAREN ( bool_unit )( SEMI ORDERED_DEF|)RPAREN SQUARE_PAREN_L ( boolsample )+ SQUARE_PAREN_R;

boolean_keywords: ( "false"| "true");
unit: "unit" DEFINITION STRING_LITERAL;
numeric_int_sample: LPAREN NUMBER SEMI NUMBER RPAREN;
numeric_real_sample: LPAREN NUMBER SEMI NUMBER RPAREN;
stringsample: LPAREN STRING_LITERAL SEMI NUMBER RPAREN;
real_pdf_sample: LPAREN NUMBER SEMI NUMBER RPAREN;
bool_unit: "unit" EQUAL "\"bool\"";
boolsample: LPAREN boolean_keywords SEMI NUMBER RPAREN;
characterisation_keywords: ( "BYTESIZE" | "STRUCTURE" | "NUMBER_OF_ELEMENTS" | TYPE" | "VALUE");
```

2.5.5.2 Lexer (EBNF)

```
mPLUS | mMINUS | mMUL | mDIV | mMOD | mPOW | mLPAREN | mRPAREN | mSEMI | mDEFINITION | mEQUAL
| mSQUARE_PAREN_L | mSQUARE_PAREN_R | mNUMBER | mNOTEQUAL | mGREATER | mLESS | mGREATEREQUAL
| mLESSEQUAL | mSTRING_LITERAL | mDOT | mID | mWS

mPLUS:'+'; mMINUS:'-'; mMUL:'*'; mDIV:'/'; mMOD:'%'; mPOW:'^'; mLPAREN:'('; mRPAREN:')';
mSEMI:';'; DEFINITION:'='; mEQUAL:'=='; mSQUARE_PAREN_L:'['; mSQUARE_PAREN_R:']';

mDIGIT:'0'..'9';
mNUMBER: ( mDIGIT )+( '.' ( mDIGIT )+ | );
mALPHA: 'a'..'z'| 'A'..'Z';

mNOTEQUAL:"<>"; mGREATER:">"; mLESS:"<"; mGREATEREQUAL:">="; mLESSEQUAL:"<=";
mSTRING_LITERAL:"\""( mALPHA | '_' )+ "\"";
mDOT: '.';
mID:( mALPHA | '_')+;              // variable ids
mWS: ( ' '| '\t'| '\r'| '\n');     // whitespace
```

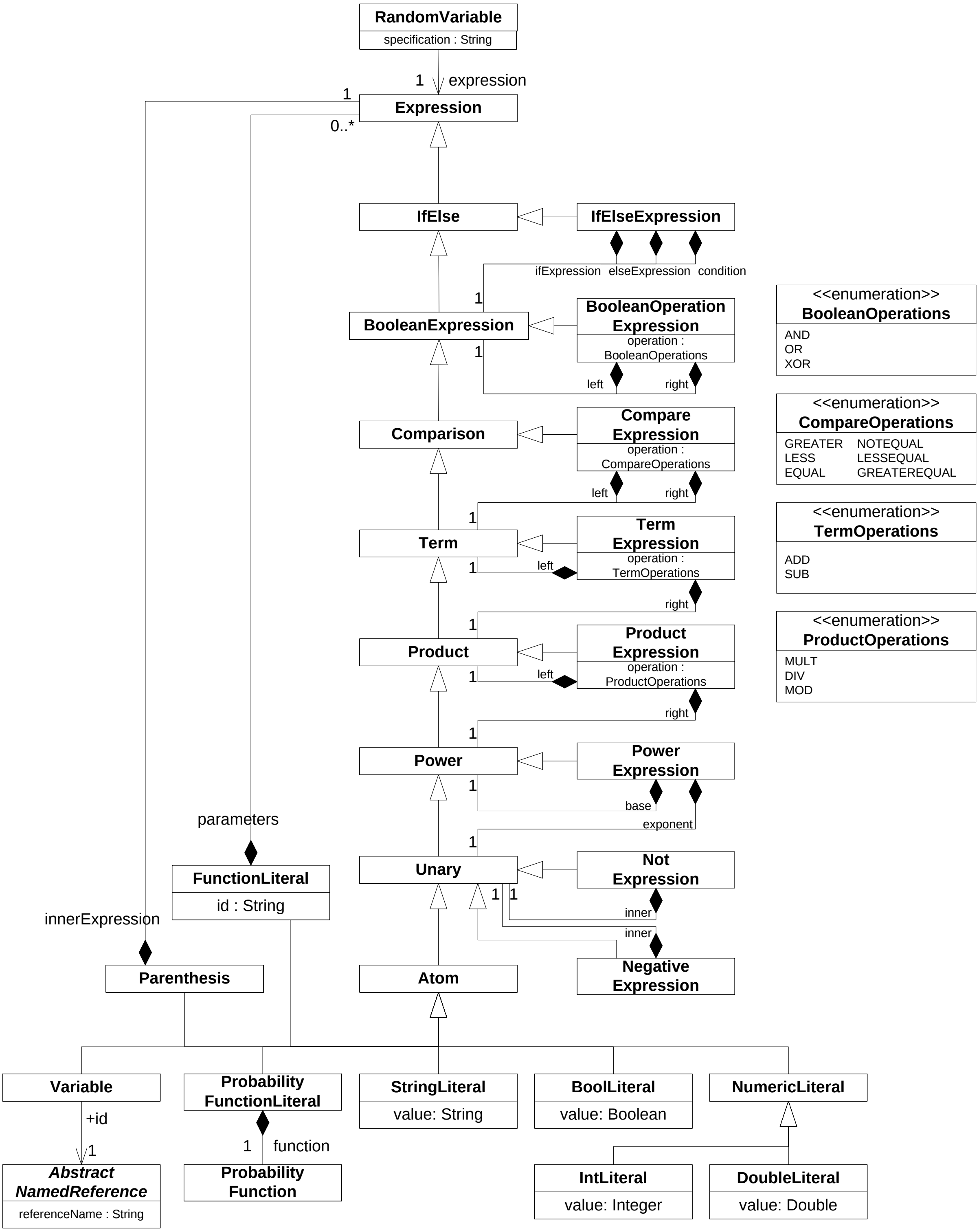


Figure 2.17: Random Variable and Stochastic Expressions Grammar (meta-model)

2.5.5.3 Examples

`DoublePDF[(1.0;0.3)(1.5;0.2)(2.0;0.5)]`

- Specifies a time interval as boxed probability density function
- the unit is seconds
- the probability of the time being between 0 and 1 second is 30 percent (0.3)
- the probability of the time being between 1 and 1.5 seconds is 20 percent (0.2)
- the probability of the time being between 1.5 and 2 seconds is 50 percent (0.5)
- the probabilty of the time being longer than 2 seconds is 0 percent.
- all probabilities sum up to 1.0

`IntPMF[(27;0.1)(28;0.2)(29;0.6)(30;0.1)]`

- Specifies the number of executing a loop as a probability mass function (PMF)
- the unit is iterations
- the probability of executing the loop exactly 27 times is 10 percent (0.1)

`DoublePMF[(22.3;0.4)(24.8;0.6)]`

- Specifies a floating point variable charcterisation as a probability mass function (PMF)
- unit is omitted
- the probability of the variable taking the value 22.3 is 40 percent (0.4)

`EnumPMF[("circle";0.2) ("rectangle";0.3)("triangle";0.5)]`

- Specifies a probability mass function over the domain of a parameter
- Graphics-Objects can either be circles, rectangles, or triangles with the respective probabilities

`BoolPMF[(false;0.3)(true;0.7)]`

- Specifies a probabilty mass function for a boolean guard on a branch transition
- The guard is false with a probability of 30 percent and true with a probablity of 70 percent.

`23`

- An integer constant
- Can be used for example for loop iteration numbers, variable characterisations or resource demands

`42.5`

- An floating point number constant
- Can be used for variable characterisations and resource demands (not for loop iterations)

`"Hello World!"`

- A string constant

`number.VALUE`

- Characterises the value of the variable "number"
- You can assign a constant or probabilty function to a characterisation
- For example, number.VALUE = 762.3 or number.VALUE = DoublePMF[(22.3;0.4)(24.8;0.6)]

`graphic.TYPE`

- Characterises the type of the variable "graphic"
- For example: graphc.TYPE = "polygon"

`file.BYTESIZE`

- Characterises the size of variable "file" in bytes

`array.NUMBER_OF_ELEMENTS`

- Characterises the number of elements in the collection variable "array"
- For example:

  ```
  array.NUMBER_OF_ELEMENTS = IntPMF [(15;0.1)(16;0.9)]
  ```

`set.STRUCTURE`

- Characterises the structure of the collection variable "set"
- For example: sorted, unsorted

```
2+4, 34.3-1, 88.2*1.2, 14/2, 60\%12
number.VALUE * 15, file.BYTESIZE / 2
```

- Arithmetic expressions can combine constants
- Allowed are + (addition), - (substraction), * (multiplication), / (division),
- Arithmetic expressions may include variable characterisations

```
DoublePDF[(1.0;0.3)(1.5;0.2)(2.0;0.5)] * 15

IntPMF[(1124.0;0.3)(1125.5;0.7)] + 2.5

DoublePDF[(12.0;0.9)(15;0.1)] -
DoublePDF[(128.0;0.3)(256;0.2)(512.0;0.5)]
```

- Arithmetic expressions can also combine probability functions

```
number.VALUE < 20, foo.NUMBER_OF_ELEMENT == 12,
blah.VALUE >= 108.3 AND fasel.TYPE == "mytype"
```

- Boolean expressions evalute to true or false
- You can use them on guarded branch transitions
- Valid operators are $>$ (greater), $<$ (less), == (equal), != (not equal), $\geq$ (greater equal), $\leq$ (less equal), `AND`, `OR`

Chapter 3

Concepts

3.1 Component Developer

3.1.1 Overview

Component developers are responsible for the implementation of software components. They take functional and extra-functional requirements for components to be developed and turn them into component specifications and executable software components. They may also receive specifications from other parties and implement components against them.

Component developers deposit their specifications and implementations into *repositories*, where they can be accessed by software architects to compose systems or by other component developers to create composite components. To provide an overview of the following section, Figure 3.1 contains an exemplary component repository, which contains most of the entities supported by the PCM.

Interfaces, components, and data types are first-class-entities in PCM repositories. They may exist own their own and do not depend on other entities. For example, Figure 3.1 contains the *interface* `MyInterface` (upper left), which is not bound to a component. The interface contains a list of service signatures. Interfaces may also contains protocol specifications, which restrict the order of calling its services, or QoS specifications, which describe their extra-functional properties. Section 3.1.2 describes interfaces in detail.

Components may provide or require interfaces. The binding between a component and an interface is called "provided role" or "required role" in the PCM. We distinguish provided and requires roles depending on the meaning of an interface for a component. For example, figure 3.1 contains the component `A` (top), which is bound to the interface `YourInterface` in a providing role. Section 3.1.3 details on the relationship between components and interfaces.

Common *data types* are needed in repositories, so that the signatures of service specifications refer to standardised types. In the PCM, data types can be primitive types, collection types, or composite types to build complex data structure. Figure 3.1 contains a `PrimitiveDataType` "INT" and a `CollectionDataType` "INT-Array", which only contains "INTs" as inner elements. Section 3.1.2 contains more information on data types.

Different types of components can be modelled in the PCM to a) reflect different development stages and b) to differentiate between basic (atomic) components and composite components.

Concerning a), Figure 3.1 contains the components `B`, which has a `ProvidedComponentType` and does not contain required interfaces, `C`, which has a `CompleteComponentType` and contains no inner structure, and component `D`, which has an `ImplementationComponentType` and may contain an inner structure. Components may be refined from `ProvidedComponentTypes` to `ImplementationComponentTypes` during design.

Concerning b), component `E` in Figure 3.1 is a composite component. The PCM is a hierarchical component model, which allows composing new components from other components. From the outside, composite components look like basic components, as they publish provided and required interfaces. Within the composite component, they use delegation connectors to forward requests to inner components and assembly connectors to bind inner components. Composite components may also include other composite components (notice component `G` within component `E`).

For each provided service, basic components may include a mapping to required services, a so-called `ServiceEffectSpecification` (SEFF). It models the order in which required services are called by the provided service and may also include resource demands for computations of the service, which are needed for performance predictions. SEFFs are an abstract behavioural description of a component designed to preserve the black-box principle. Section 3.1.4 explains different types of SEFFs and their application.

3.1.2 Interfaces

Szyperski et al. emphasise the relation between components and interfaces: “Interfaces are the means by which components connect [6, p. 50].” For components, interfaces are a key concept serving multiple purposes. First, this section will describe the structure of PCM interfaces and then discuss the role of interfaces as contracts as well as inheritance of interfaces.

An interface within the PCM consists of a list of sigatures (mandatory), a protocol specification (optional). The following explains both concepts in more detail.

3.1.2.1 Signatures

A signature in the PCM is comparable to a method signature in programming languages like C# or Java. It is widely compatible with the OMG’s IDL standard [25, p. 3-1 and following]. Each signature of an interface is unique and contains:

- A type of the return value or `void` (no return value)
- An identifier naming the service
- An ordered set of parameters (0..*). Each parameter is a tuple of a datatype and an identifier (which is unique across the parameters). Additionally, the modifiers `in`, `out`, and `inout` (with its OMG IDL semantics, cf. [25, Chapter 3]) can be used for parameters.
- An unordered set of exceptions.

A signature has to be unique for an interface through the tuple (identifier, parameters). An interface has a list of 1..* signatures and a signature is assigned to exactly one interface. However, different interfaces can define equally named signatures, which are distinguished by their parameters. If, for example, `void doIt()` is defined for interface `A` and `B`, `void doIt()` is not identical in both interfaces.

3.1.2.2 Protocols

A protocol is a set of call sequences to the services of a single interface and can be optionally added to an interface specification. In general, a protocol defines the set of all possible call sequences of the interface’s signatures. Depending on the role of the interface (cf. Section 3.1.2.5), protocols are interpreted differently. Protocols of provided interfaces specify the order in which services have to be called by clients. Protocols of required interfaces specify the set of all possible call sequences to required services. However, the specification of a protocol is independent of its interface’s role.

Figure 3.2 shows an example of protocols visualised as finite state machine. Nodes represent states, while edges represent calls to services and are labelled with signatures. The figure on the left hand side shows the protocol for the interface `IReaderWriter` as a finite state machine. First, `open(..)` is called. Then, `read(..)` and `write(..)` can be called in an arbitrary sequence. Finally, `close()` terminates the protocol.

Besides finite state machines, different *formalisms* can be used to model protocols. For example, Petri nets or stochastic process algebras could model interface protocols. However, the choice of a formalism implies the possible analyses. For example, to check the interoperability of two components, language inclusion has to be checked. The language inclusion is undecidable for Petri nets in the general case, so protocols modelled with Petri nets cannot be checked for interoperability.

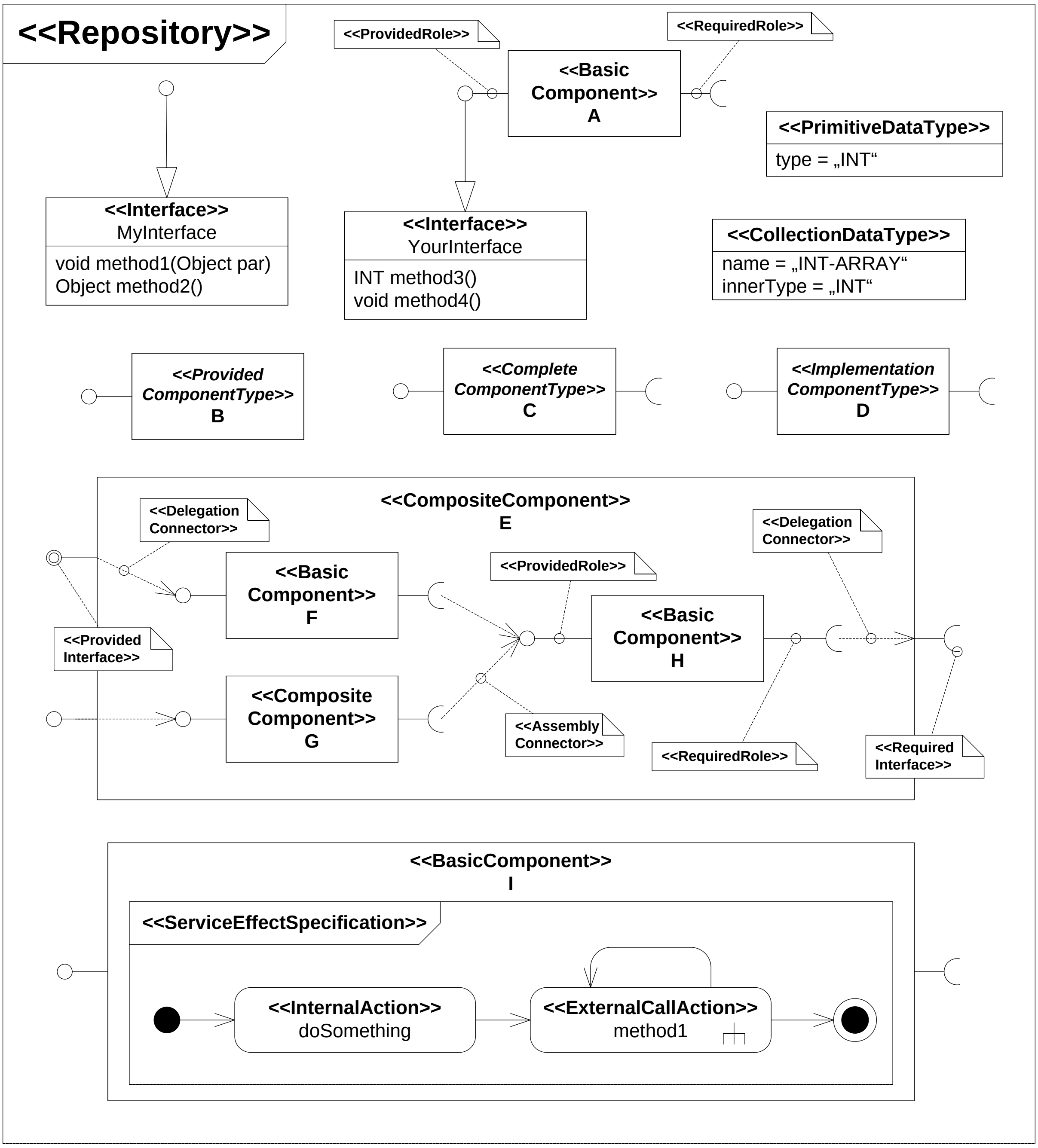


Figure 3.1: Repository Example

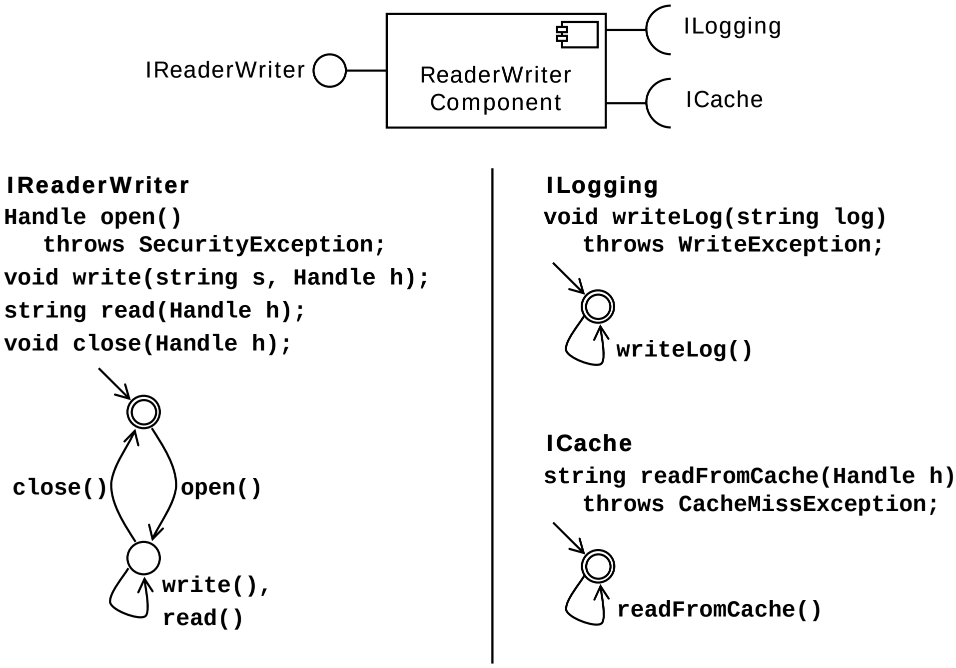


Figure 3.2: Example: Interfaces with Signature Lists and FSM protocols

3.1.2.3 Interfaces as Contracts

Interfaces are applied to specify the allowed communication between components. The contracts specified in the interface (method contracts, invariants) characterize the valid behaviour of these entities. In object-oriented languages an object can act in two roles with respect to an interface: server or client. In the server role, the object "implements" or "realizes" the operations specified in the interfaces and observes the method pre- and postconditions. In the client role, the object calls services offered in a given interface by fulfilling the precondition and expecting the postcondition. However, in both cases the interface and its associated contracts serve both roles as contract on which they can rely.

As with legal contracts, interfaces can exist even if no one actually declared their commitment to them, i.e., there is no specific client or server. For example, this is used to define a certain set of standardised interfaces of a library to enable the construction of clients and servers of these libraries independently. Thus, in our model the concept *Interface* exists as first class entity which can be specified independent from other entities.

An interface protocol is a special case of the more general concept of arbitrary preconditions for methods. Any kind of protocol can be expressed via preconditions. Thus, the protocol is an abstraction of the set of all preconditions. The abstraction is often based on the expressiveness of the used specification formalism.

3.1.2.4 Interface Inheritance

The subtype relationship of any two arbitrary interfaces I_1, I_2 can be specified as follows. Interface I_1 is subtype of I_2 if it is able to fulfil at least the contracts of I_2. In detail, this means it has to be able to handle all the (single) method calls which I_2 can handle. Additionally, it must also at least support the call sequences which I_2 supports. A common constraint for the hierarchy of interfaces is that any given interface can not be supertype of itself, which gives us a acyclic subtype hierarchy.

This definition of interface inheritance is required to support contra-variance – cases in which not the original interface is used, but a super- or sub-type. A sub-type can replace a super-type at the provided side of a component, while a super-type can be used instead of a sub-type at the required side.

3.1.2.5 Roles

Components use interfaces to declare their provided and required functionality. These interfaces are often referred to as provided and required interfaces. Since interfaces themselves can be considered as contracts that do not make any statement about the participants of the contract (cf. Section 3.1.2), the role of the component for the contract needs to be set elsewhere.

The PCM uses `Roles` for this purpose. A `Role` associates an interface to a component. The type of the association determines whether the component offers or demands the interfaces. `ProvidedRoles` reference the interfaces offered by a component. In this case the component takes the role of a server. It implements the services defined in the interfaces. Furthermore, it can rely on the call sequences defined in the interface's protocol, because its clients will adhere to it. On the other hand, `RequiredRoles` reference the interfaces requested by a component. The component uses the interfaces to implement its functionality. It will only call the its services according to the interface's protocol specification.

3.1.3 Components

To create a software system, software architects can use existing components from repositories or specify new ones. So, some of the components in an architecture are already specified while others are only sketched. As a consequence, we cannot characterise component-based development processes into the classical top-down (i.e., going from requirements to implementation) or bottom-up (i.e., assembling existing component to create an application) categories. Instead, it is a mixture of both approaches.

This mixture needs to be reflected in the component model, since software architects must be able to use fully specified components in combination with nearly unspecified ones in their architectural descriptions. The PCM reflects this requirement by a so-called component type hierarchy. It distinguishes three abstraction levels for software component specifications (from abstract to concrete): provided types, complete types, and implementation types. On the most abstract level, *provided types* specify a component's provided interfaces leaving its requirements and implementation open. This allows software architects to create ideas of components, leaving their realisation unspecified. On the middle level, *complete types* fully specify a component's required and provided interfaces, but do not make any statements about its internal structure. This is more concrete than provided types as the component's dependencies have already been defined. However, the actual implementation (how provided services use required ones) still remains open. Software architects can use complete types for substitution of one component by another, if they have a selection of multiple components (e.g., different variants or versions) with the same functionality but different extra-functional properties. Last but not least, *implementation types* abstractly specify the internal behaviour of a software component. Their behavioural model describes how the provided services of the actual component implementation call its required services. Behavioural descriptions of software components are needed to evaluate extra-functional properties such as reliability or performance of software architectures.

This section provides an overview on the different component types and their relationships. It first introduces the ideas and concepts of the type hierarchy. Then, the realisation in the PCM and the meta model of the type hierarchy are explained. The following describes the concepts of the three levels of the type hierarchy. The explanation starts with the most concrete implementation type, since it conforms to the intuitive understanding of a software component for most developers. Based on the concepts introduced the more abstract concepts of complete and provided types are explained.

3.1.3.1 Implementation Component Type

Implementation (component) types include descriptions of a component's provided and required interfaces as well as abstract specifications of its internal structure. The specification of the internal structure depends on the way the component is realised. In general, components can either be implemented from scratch or composed out of other components. In the first case, the implemented behaviour of each provided service needs to be specified with a service effect specification (SEFF, cf. Section 3.1.4) to describe the component's abstract internal structure. We refer to such components as *basic components*, since they form the basic building blocks of a software architecture. On the other hand, developers can use existing components to assemble new, *composite components*. The internal structure of these components is the structure of the assembly (i.e., the included components and their interconnections). The following explains the concepts of basic and composite components in more detail

Basic Components Basic components are atomic building blocks of a software architecture. They cannot be further subdivided into smaller components.

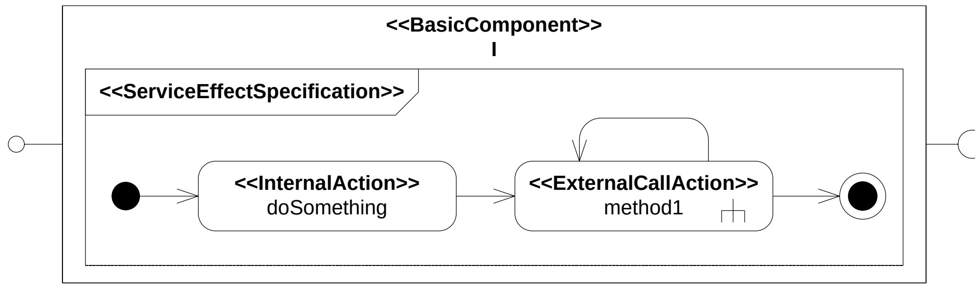


Figure 3.3: Example of a Basic Component.

Basic components encapsulate their internal structure (black box view). For reasoning about a basic component's properties, it may contain SEFFs, which describe the dependency between provided and required roles (cf. Figure 3.3). SEFFs abstract from the component's internal behaviour and only reveal necessary internals to reason on the component properties, such as protocol interoperability and QoS. Section 3.1.4 describes SEFFs in detail.

Composite Components Composite components are created by assembling other, existing components (cf. Figure 3.4). They base on composed structures (cf. Section 2.2.2), which contain a set of assembly contexts, delegation connectors, and assembly connectors. Assembly contexts embed components into the composite component. Assembly connectors bind required and provided roles of inner components in different contexts. Delegation connectors bind provided (required) roles of the composite component with provided (required) roles of its inner components.

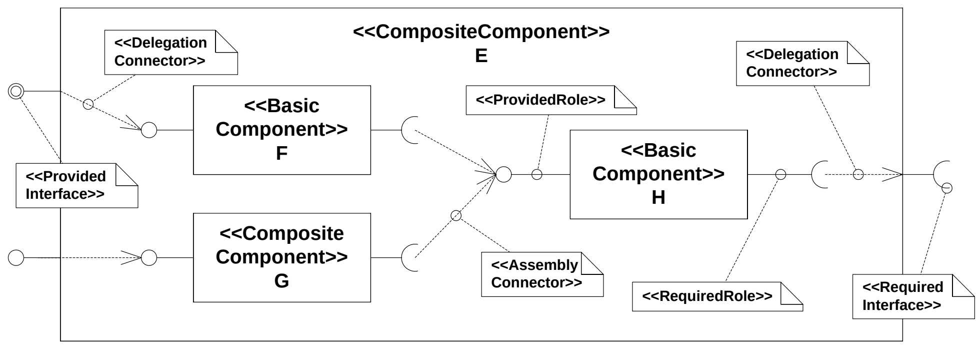


Figure 3.4: Example of a Composite Component.

Composite components may contain other composite components, which again are composed of other components. This enables building arbitrary hierarchies of nested components. Like basic components, composite components may contain SEFFs. However, these SEFFs are not specified manually by the component developer, but can be computed by combining the SEFFs of the inner components.

3.1.3.2 Complete Component Type

Complete (Component) types abstract from the realisation of components. They only have provided and required roles omitting the components' internal structure (i.e., the service effect specifications or encapsulated components). Thus, complete types represent a black box view on components.

Software architects can integrate complete types into their architectures, which are fully connected with their provided and required roles. However, as their internal structures are not specified, they can be substituted by basic or composite components in later development stages. This is especially useful if the component is developed in a top down fashion. As described in Section 2.1.2, software architects can use complete types as a requirement specifications handed over to third parties. Component developers provide the actual implementations and specifications which complete the software architecture as soon as they are available.

If a component's implementation and specification does not exist, software architects can still model and evaluate an architecture. However, they have to provide basic QoS estimates for the complete component types in their architecture to evaluate its QoS attributes. Furthermore, the QoS results cannot be expected to be as accurate as for implementation component types.

3.1.3.3 Provided Component Type

Provided (Component) types abstract a component to its provided interfaces, leaving its requirements and implementation details open. So, provided types subsume components which offer the same functionality, but with different implementations and requirements. As different implementations might require different services from the environment, provided types omit required interfaces. Provided types allow software architects to focus on a component's functionality.

Using provided types, software architects can draft ideas on how functionality can be partitioned among different components without worrying about their implementation. In the initial phases of architectural design, it often does not make sense to arrange all details of a component, since most of them depend on the actual implementation and thus need to be specified by component developers. As during this phase the actual implementation is unknown, also the required interfaces of a component cannot be stated. However, software architects can still pre-evaluate software architectures containing provided-types by giving basic QoS estimates for them. This gives rough estimates about the quality of a software system and defines QoS requirements for the component implementation.

3.1.3.4 Type Hierarchy

The provided, complete, and implementation component types can be organised in a hierarchy as shown in Figure 3.5. Provided types are on the top, most abstract level, since they only specify provided roles. The lower levels extend provided types with requirement and implementation specifications. On the middle level, complete types extend provided types with required roles, but still abstract from the actual implementation of a component. Implementation types on the lowest level of the hierarchy can either be composite or basic components. Thus, they specify the provided and required roles as well as the abstract internal structure of a component.

The different levels of the hierarchy are related to one another by the *conforms* and *impl-conforms* relationships. These relationships define under which conditions a component specification on a lower level is of a higher level type. A complete type *conforms* to a provided type if it offers at least the functionality specified in the provided type. Furthermore, an implementation type *impl-conforms* to a complete type if it offers at least the functionality of the complete type and requires at most the functionality of it required interfaces. The following explains both relationships in more detail and introduces a notion of substitutability based on their definition.

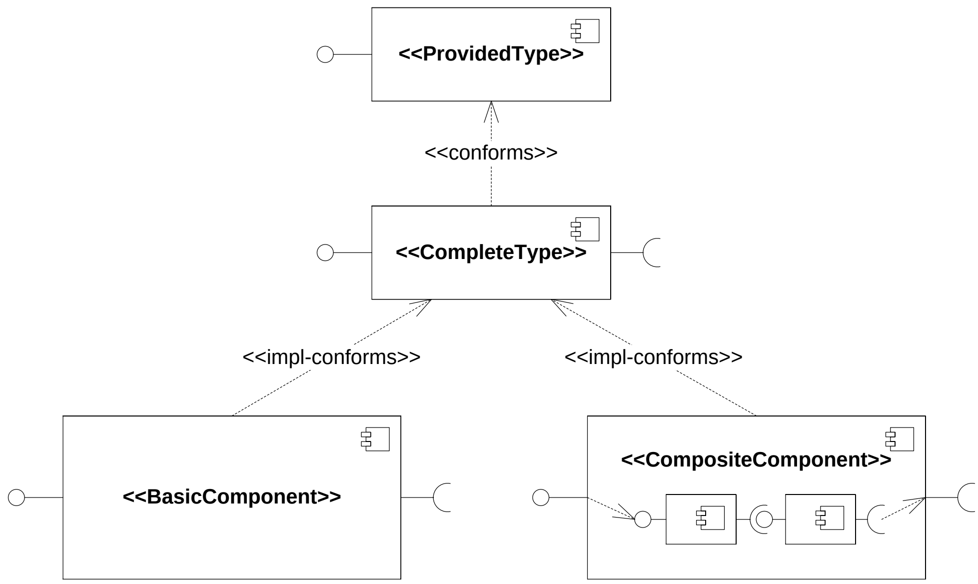


Figure 3.5: Component Type Hierarchy.

Conforms Relation The conforms relation is a subtype relation of provided and complete types. Abstractly speaking, a complete type conforms to provided type if it provides at least the functionality specified in the provided type. In order to concretise this statement, we need to define the provided functionality of a component and its relation.

Provided roles of components define their offered functionality associating interfaces to components. Thus, the conforms relation is defined on the provided interfaces of components [1].

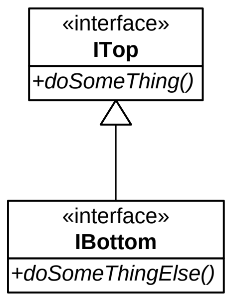


Figure 3.6: Example of interface inheritance.

Section 2.2 introduces the concepts of interfaces in the PCM. At this point, we only give a brief overview on the fundamental concepts. Interfaces are organised in an inheritance hierarchy. So, an interface can have multiple supertypes and subtypes. Basically, a supertype of an interface provides less and the subtype provides more services. Figure 3.6 shows an example. There, interface `ITop` is a

[1] Another option would be the definition of the conforms relation on services provided by a component neglecting the corresponding interfaces. However, this is ambiguous, since two interfaces can provide syntactically equal services, but with different semantics. Furthermore, the PCM allows the specification of protocols for interfaces, which have to be considered in the conforms relation.

supertype of interface `IBottom` while `IBottom` is a subtype of `ITop`.

In order to give a meaningful definition of the conforms relation, we have to consider the inheritance hierarchy of interfaces. We can refine the definition of the conforms relation.

A complete type conforms to a provided type if it provides at least the interfaces or subtypes of the interfaces specified in the provided type. More formally, let *Prov* be the set of provided interfaces of a component type including all supertypes. Then a complete type `C` conforms to a provides type `P` if $Prov_P \subseteq Prov_C$, the interfaces provided by `P` are a subset of the interfaces provided by `C`.

Implementation-Conforms Relation The impl-conforms relation is a subtype relation between implementation types and complete types. Abstractly speaking, an implementation type (either a basic or composite component) conforms to a complete type if it provides the same or more functionality and requires the same or less functionality than the complete type.

With respect to the provided functionality, the impl-conforms relation is similar to the conforms relation. In addition, the required functionality of an implementation type must be less or equal to the required functionality of a complete type. Analogously to the provided functionality, the required functionality is specified in the components' required roles (i.e., the interfaces associated to the component with its required roles). Considering the supertype and subtype relation of interfaces, we can define the impl-conforms relation as follows.

An implementation type conforms to a complete type if it provides at least the interfaces or subtypes of the interfaces provided by the complete type and if it requires at most the interfaces or supertypes of the interfaces required by the complete type. More formally, let *Prov* be the set of provided interfaces of a component type including all supertypes and *Req* be the set of required interfaces of a component type including all subtypes. Then an implementation type `I` conforms to a complete type `C` if $Prov_C \subseteq Prov_I$ and $Req_C \supseteq Req_I$, the interfaces provided by `C` are a subset of the interfaces provided by `I` and the interfaces required by `C` are a superset of the interfaces required by `I`.

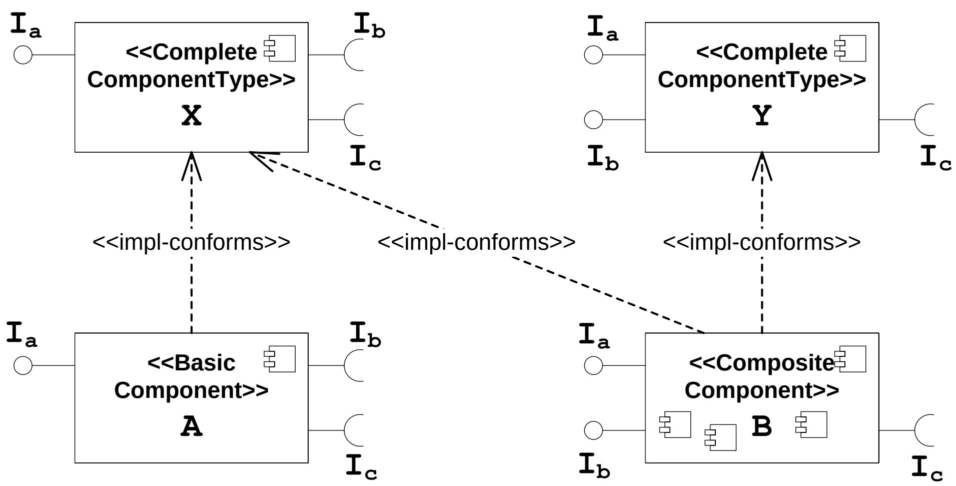


Figure 3.7: Example for Component Type Conformance.

Figure 3.7 shows an example for conforms relations of implementation and complete components. Complete component `X` provides interface `Ia` and requires `Ib` and `Ic` while complete component `Y` provides `Ia` and `Ib` and requires `Ic`. Basic component `A` provides and requires the same interfaces as complete type X and thus impl-conforms to X as indicated by the dashed arrow with the stereotype «impl-conforms». However, A does not conform to Y since it does not provide interface Ib and additionally requires it. Composite component B impl-conforms to both types X and Y as it provides interfaces Ia and Ib and only requires Ic.

Cardinality of the Conforms Relations The conforms as well as the impl-conforms relations are many-to-many relations between two levels of the component type hierarchy. Each implementation type can conform to multiple complete types and each complete type can be implemented multiple times. Figure 3.7 illustrates this. Composite component B impl-conforms to complete types X and Y and complete type X is implemented by basic component A and composite component B. The same holds for the conforms relation as well. Each provided type can abstract multiple complete types and each complete type can conform to multiple provided types.

Substitutability The main application of both conforms relations is the definition of substitutability for software components in the PCM. A component can substitute another component if it conforms to its type. Depending on the type of conforms relation, the substitution of a software component can have different effects. The following discusses this in more detail.

Assume we have a software architecture where an implementation type A is used. If a component B shall substitute A and B conforms to the provided type of A, but not impl-conforms to its complete type, B provides at least the interfaces offered by A, but requires additional interfaces. Thus, replacing A by B in the given architectures can lead to problems since not all of its requirements are fulfilled. On the other hand, a component B' that impl-conforms to A can easily replace A, since it provides the necessary interfaces and all its requirements can be fulfilled by the surrounding architecture.

3.1.3.5 Type Hierarchy Meta Model

A part of the meta model describing the component type hierarchy is analogous to the structure of the type-hierarchy itself (cf. Figure 3.8). Each type level is a specialisation of the upper levels. So, lower levels in the hierarchy only add information to the component specification, e.g. the complete type adds mandatory required roles. Thus, lower type levels inherit the attributes of the upper levels.

However, the inheritance between the different type levels is only partially connected to the conforms relations. As a consequence of the inheritance, an instance of a basic component is as well an implementation, complete, and provides type. Due to the definition of the conforms relation, it certainly conforms to itself. However, the conforms relation is not restricted to itself, it can conform to other component types as well.

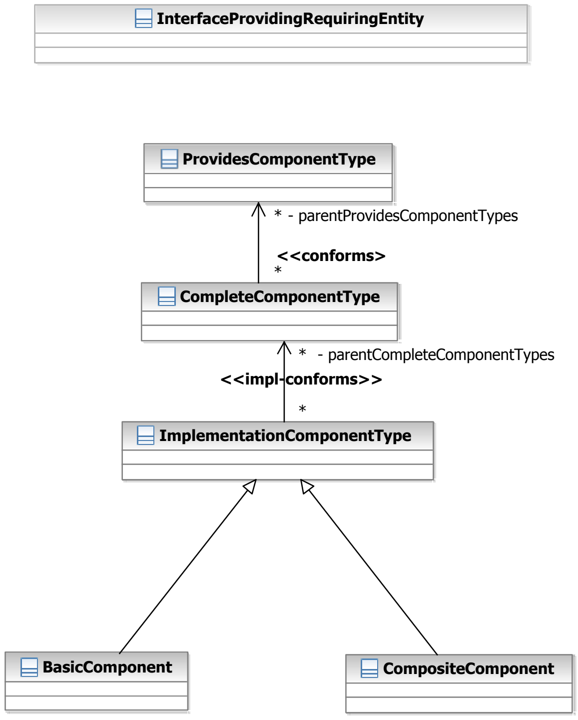


Figure 3.8: Meta Model of the Component Type Hierarchy.

3.1.4 Service Effect Specification

3.1.4.1 Motivation

The goal of the PCM is to provide modeling capabilities that enable QoS analyses of component-based software architectures. As clients perceive different QoS characteristics of a provided service in a component-based architecture depending on a particular context, component developers have to provide parameterised specifications of the QoS attributes of their components. Such context dependencies for a specific component service may originate from (a) input parameters (including the current component internal state), (b) resource usage, and (c) usage of required services. These influences have to be made explicit in the service's specification.

To achieve accurate QoS analyses, a description of the usage of required services (influence (c)) for each provided service of a component is useful, because the QoS characteristics perceived at the provided interface can depend on QoS characteristics of calls to required services. For example, consider a provided service calling a slow required service. In this case, the response time of the provided service will be perceived as slow by its clients, because the execution time of the slow required service has to be included in its own execution time (details can be found in [26]). Software architects cannot know how requests to a provided service of a component are propagated to required services if no dependencies between them are specified. Thus, component developers have to enhance their component specifications with a description of such intra-component dependencies to enable accurate specification-based QoS analyses by third parties.

In the following, service effect specifications (SEFFs) are discussed focussing on influence (b) and (c).

The influence of input parameters (influence (a)) is discussed separately in Section 3.1.5.

3.1.4.2 Resource Demanding Service Effect Specification

A *resource demanding service effect specification* (RDSEFF) [1] is a special type of SEFF designed for performance and reliability predictions. Besides dependencies between provided and required services of a component, it additionally includes notions of resource usage, data flow, and parametric dependencies for more accurate predictions. Its control flow is hierarchically structured and can be enhanced with transition probabilities on branches and numbers of iterations on loops. In the following, the meta model of the RDSEFF will be illustrated, and its design rationale will be explained. For understanding and clarity, the illustration of the meta model and the concept descriptions are spread over several paragraphs.

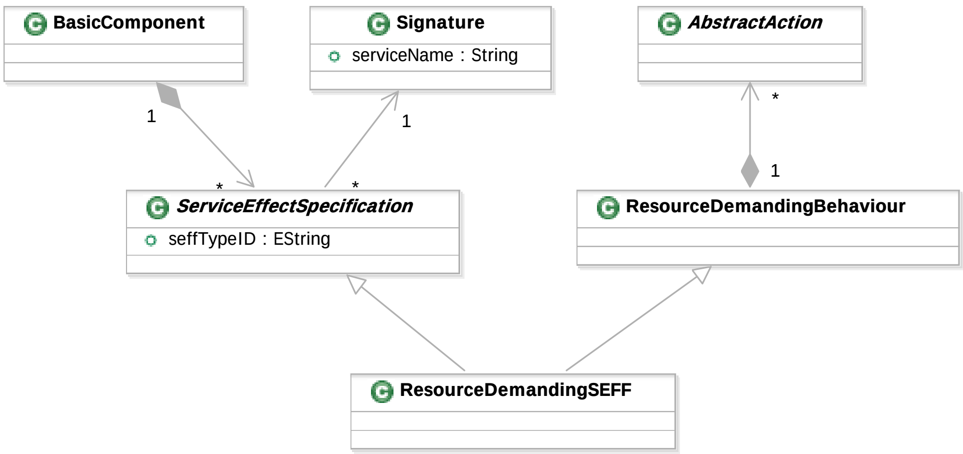


Figure 3.9: Overview of the RDSEFF

Overview Figure 3.9 shows how RDSEFFs are connected to the PCM and contains their main parts. Each `BasicComponent` can contain a number of `ServiceEffectSpecifications`, each of which references a signature of a provided service of the component. Each provided service can be described with different types of SEFFs.

A `ResourceDemandingSEFF` is a `ServiceEffectSpecification` and a `ResourceDemanding-Behaviour` at the same time inheriting from both classes. The reason for this construct lies in the fact, that `ResourceDemandingBehaviours` can be used recursively inside themselves to describe loop bodies or branched behaviours (explained later), and these inner behaviours should not be RDSEFFs themselves.

The `ResourceDemandingBehaviour` is designed to reflect different influence factors on the performance and reliability of a component service. It contains a set of `AbstractActions` to model

- calls to required services,
- resource usage by internal activities, and the
- corresponding control flow between required service calls and resource usage.

Resource Demand To conduct QoS analyses, component specifications must contain information on how system resources, such as hardware devices or middleware entities are used by components. Ideally,

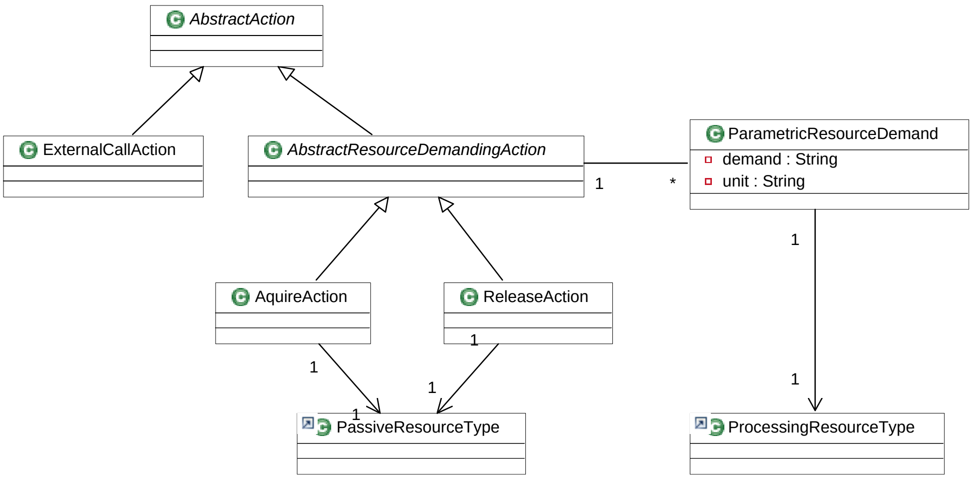


Figure 3.10: Resource Usage in RDSEFFs

component developers would specify a timing value for the execution time of each provided service of a component. However, these timing values would be useless for third party users of the component, because they would depend on the specific usage profile, hardware environment, software platform, and attached required services the component developer had used while measuring them.

Thus, component developers have to specify the *demand* each provided service places on resources instead of a timing value. Other than a timing value, the demand is independent from concrete resources. For example, a component developer could specify the number of CPU cycles of a specific operation within a service or the number of bytes read from or written to an I/O device. These demands have to be specified against abstract resource types, because the component developer does not know all possible resources the component could be deployed on. Only software architects and deployers know the concrete resources the component shall be used on and can define a specific deployment context (i.e., a resource environment model, Section 3.3.4). With this concrete context, for example, the execution time of one CPU cycle or the time to read one byte from an I/O device is specified. Then, actual timing values can be derived from the resource demands.

These considerations have been mapped to the meta model of the RDSEFF (see Figure 3.10). `AbstractActions` can either be external calls (`ExternalCallAction`), which reference required services and do not produce resource demands themselves, or internal computations actions (`AbstractResourceDemandingActions`), which actually place demands on resources. These `ParametericResourceDemands` contain a demand (e.g., “127”) and a unit (e.g., “bytes”). The demand can also be specified in dependency to the service’s input parameters (cf. Section 3.1.5.1).

Resource demands reference `ProcessingResourceTypes` from the `ResourceType` package of the PCM (Section 3.3.3). Once the concrete processing resource, such as a CPU or network device, is specified, the actual resource demands can be placed on them to calculate timing values.

Besides active resources, such as CPUs, I/O devices, storage devices, memory etc., component service may also acquire or release *passive resources*, such as semaphores, threads, monitors, etc. These resources usually exist in a limited number, and a service can only continue its execution if at least one of them is available. Passive resource are themselves not able to process requests and do not allow to place demands on them. They can only be acquired and released, which can be modelled with the `AquireAction` and `ReleaseAction` (see Figure 3.10). These actions reference `PassiveResource-`

Types from the resource type package of the PCM.

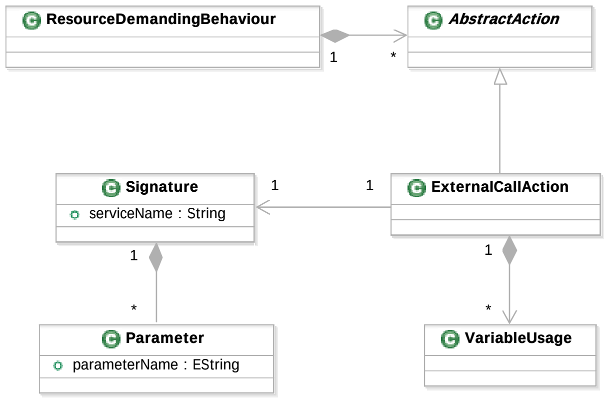


Figure 3.11: External Service Calls and Parameter Usage in RDSEFFs

External Calls and Parameter Usage RDSEFFs allow the specification of calls to *required services* with `ExternalCallActions`, which are themselves `AbstractActions` (see Figure 3.11), but produce no resource demands directly. The resource demand produced by executing a required service has to be specified in the RDSEFF of that service. `ExternalCallActions` reference the signature of a required service.

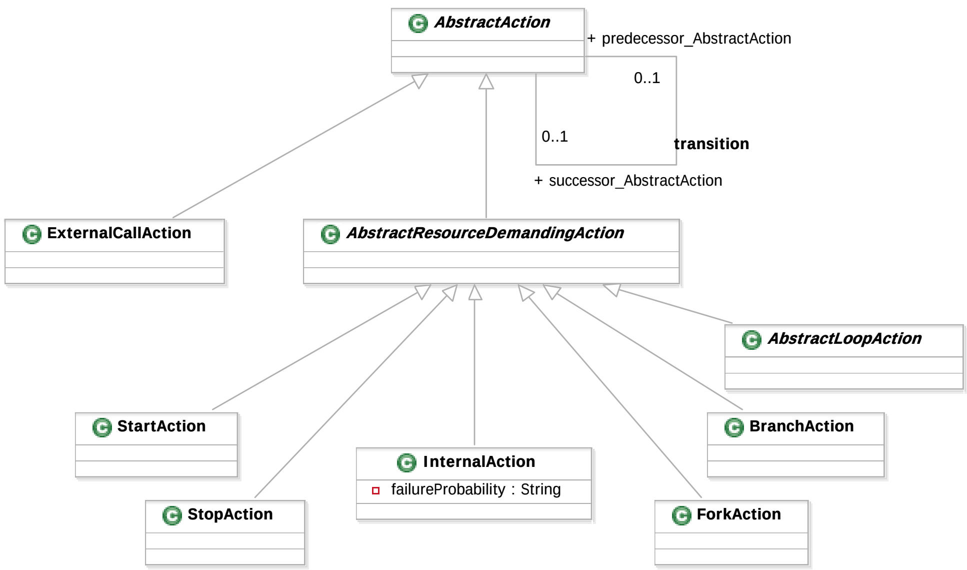


Figure 3.12: Control Flow in RDSEFFs

Control Flow RDSEFFs contain additional constructs for modeling control flow of the dependencies between provided and required interfaces (Figure 3.12). All control flow constructs are aligned in a hierarchical fashion that avoids ambiguities and eases analyses (an example will follow). A `ResourceDemandingBehaviour` contains a chain of `AbstractActions`, which each reference at most a single predecessor and successor. The first element of the chain is a `StartAction`, which has no predecessor, while the last element of the chain is a `StopAction`, which has no successor.

`InternalActions` should be used to reference `ParametricResourceDemands` for activities inside the described service, between calls to required services. In the future, they could be used to characterise the inner resource demand of *basic components* more detailed.

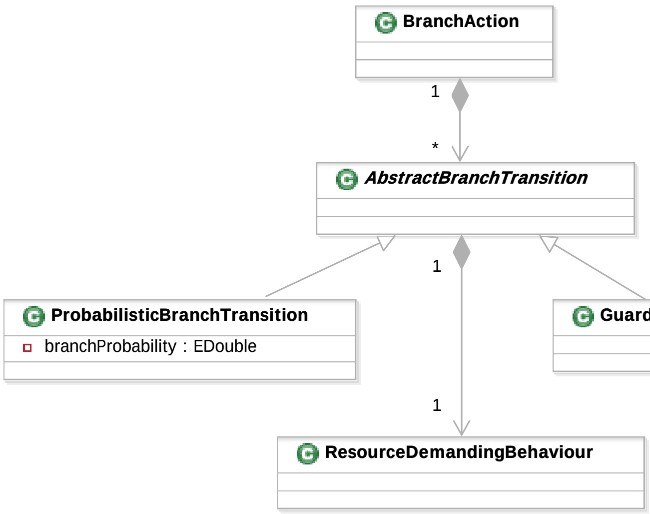


Figure 3.13: Branches in RDSEFFs

`BranchActions` split the control flow with an XOR-semantic: Exactly one of the attached `AbstractBranchTransitions` is taken when such an action is executed. Branches may result from `if`/ `then`/ `else` or case statement of the underlying source code.

Branch transitions can be either guarded or probabilistic (Fig. 3.13). `GuardedBranchTransitions`, contain a branch condition as a random variable. For example, a branch condition could be connected to the value of an input parameter ("$x < 1$"), in which case a branching probability could be computed once the value of the input parameter is known (cf. Section 3.1.5.1). `ProbabilisticBranchTransitions` directly contain a probability and not a branch condition. They can be used in case a component developer cannot specify a guard related to input parameters or just to ease the analyses.

Additionally, each type of branch transition contains a `ResourceDemandingBehaviour` to model the inner actions of the branch. Using inner behaviours avoids the need to have a merge action to join branches. Furthermore, it prevents problems, which might arise when a nested "else"-branch cannot be associated unambigiously with an according "if"-branch.

`ForkActions` split the control flow with an AND-semantic: Each of the inner forked `ResourceDemandingBehaviours` has to be executed (possibly concurrently) before the control flow continues with the successor of the corresponding `ForkAction`. Forks may for example result from the invocations of threads. Inner forked behaviours can be specified directly, or within a `SynchronizationPoint`. For behaviours specified inside a `SynchronizationPoint`, the control flow waits until the inner behaviours are completed. This allows to model synchronization barriers of concurrent control flows.

`AbstractLoopActions`, similar to `BranchTransitions` and `ForkActions`, contain inner `Resource-`

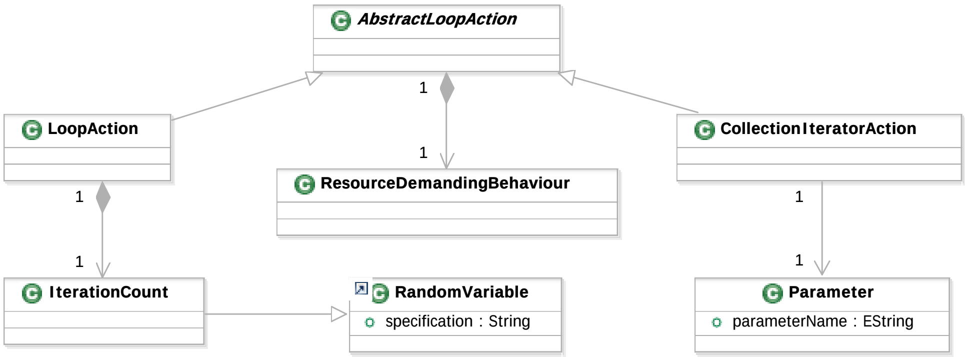


Figure 3.14: Loops in RDSEFFs

`DemandingBehaviour`, which include actions carried out in the loop body (Fig. 3.14). Loops can originate from `for` or `while` statements of the underlying source code.

Concrete loop action can be modelled either with `LoopActions` or `CollectionIteratorActions`. The former contains the number of iterations, the latter enables modelling the special but common case of iterating over a collection. See Section 3.1.5.1 for more details on specifying that the number of iterations depends on input parameters.

Modelling loops with *inner behaviours* instead of allowing cyclic references in the chain of `AbstractActions` has several advantages [27]. In Markov models, loops are specified with cycles, so that there is an entrance probability for each loop and an exit probability. The probability of entering the loop decreases if the number of loop iterations is increased. For example, entering a loop with a entrance probability of 0.9, leads to a probability of 0.81 for two loop iterations, and a probability of 0.729 for three loop iterations. Thus, the number of loop iterations is always limited to a geometrical distribution, which does not resemble practical situations well. Fixed number of loop iterations can only be specified by unrolling the loop to a number of states in Markov models. With the approach described above, it is possible to attach an arbitrary distribution function for the number of iterations to each loop.

Figure 3.15 shows a simplified example instance of an RDSEFF, which highlights the control flow concepts introduced before. Note that the constructs are hierarchically structured. Analysis algorithms can easily traverse the abstract syntax tree to make model transformations or QoS predictions.

Passive and Processing Resources Resources are divided into processing and passive resources, whose concepts are elaborated in the following.

Active resources are those which perform tasks on their own and thus can actively execute a task. This includes CPUs, hard disks, and network connections. As these resources always do some kind of job processing, we call them processing resources.

Passive resources on the other hand can be owned by a process or thread for a certain period of time. A passive resource has to be acquired to be accessed. Since passive resources can be limited, processes or threads might have to wait until a resource becomes available. Typical examples of passive resources are connection pools and thread pools. The acquisition and the release of a passive resource has to be represented in the SEFFs, which describe the control flow of a component (see page 48). If a component requires access to a limited resource, it first has to acquire it using the `AcquireAction`. After it has finished its operation, it has to release the resource using the `ReleaseAction`. The semantics of a passive resource with its `AcquireActions` and `ReleaseActions` is based on the semantics of semaphores.

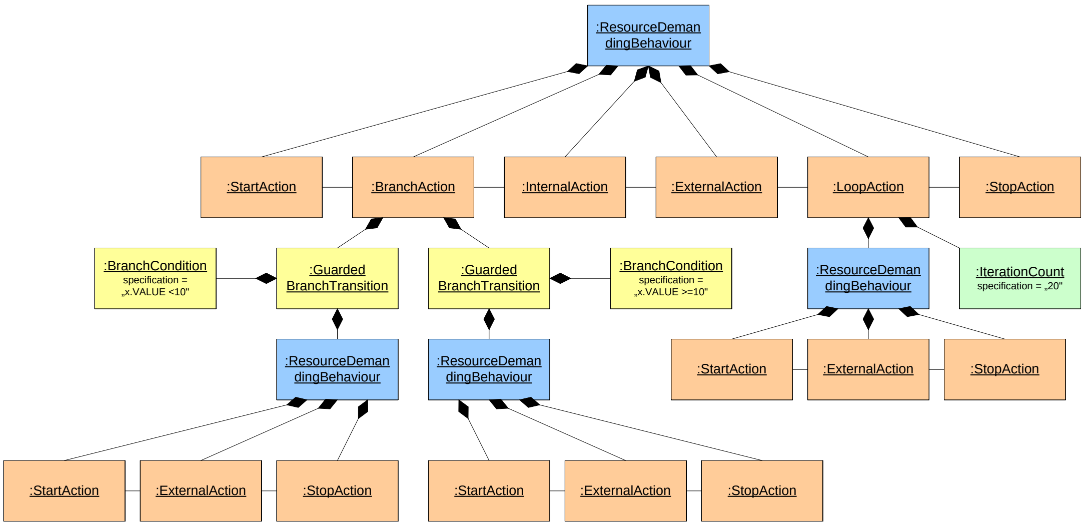


Figure 3.15: Example Instance RDSEFF highlighting control flow concepts

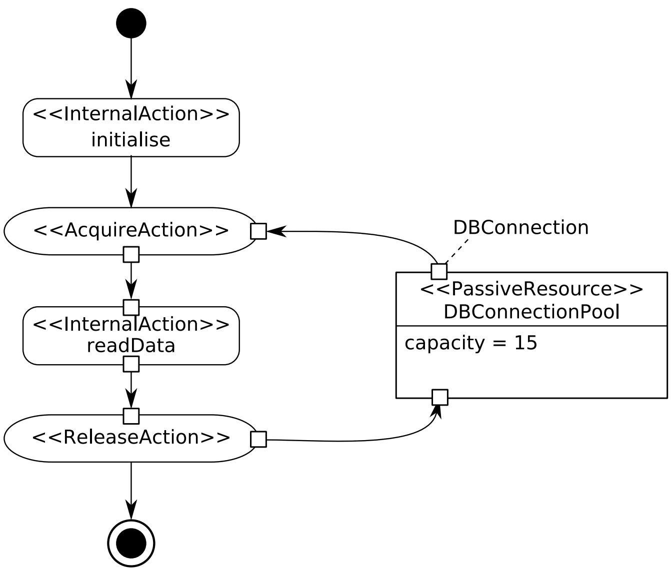


Figure 3.16: Example of a SEFF using a passive resource.

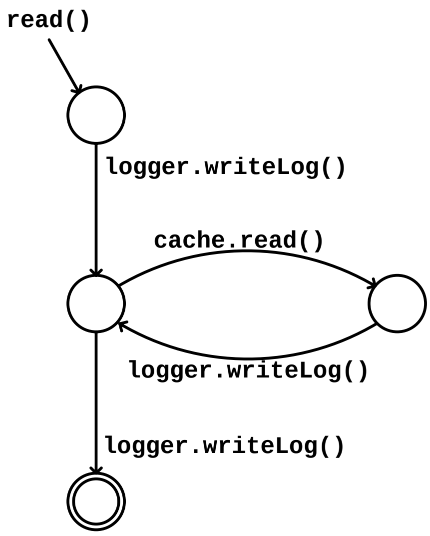
Example 3.1 (Passive Resource). Figure 3.16 shows a simple SEFF that uses a passive resource. First, the SEFF performs some initialising actions that are captured in the `InternalAction initialise`. Next, an `AcquireAction` is invoked to get a connection to the database. The `capacity` attribute of the `DBConnectionPool` indicates that there are 15 connections to the database available. If no connection is left, the `AcquireAction` blocks the current thread until a database connection is returned to the pool. The `DBConnection` object is then passed by the `AcquireAction` to the `InternalAction readData`, which reads some entries from the database. Finally, the `ReleaseAction` returns the connection object to the `DBConnectionPool` allowing other processes to use it.

Example 3.1 shows how a passive resource is used by a SEFF. The object received from the `DB-`

`ConnectionPool` is passed from one action to another. Within the actions, the object can be used. So, a passive resource can be owned and used by a process or thread for a certain period. Opposed to that, active resources cannot be owned. A scheduler decides which thread or process will be handled next by a processing resource.

3.1.4.3 Modeling SEFFs as Finite State Machines

A service effect specification (SEFF) describes how a provided service of a component calls its required services and is thus an abstraction of the control flow through the component. In the simplest case, a SEFF of a provided service is a list of signatures of the services in the component's required interfaces. For more sophisticated analyses, a SEFF can be modelled as a finite state machine (FSM), which captures sequences, branches, and loops. In any case, a SEFF captures the externally visible behaviour of a provided service while hiding its internal computations.



(a)

```
ILogging logger;
ICache cache;

[…]

public void read(Handle h){
  if (h==null){
    h = new Handle(FILENAME);
  }

  logger.writeLog(„start reading cache“);

  while(h.hasNext()){
    cache.read(h.Current());
    logger.writeLog(„accessing cache“);
  }

  logger.writeLog(„end reading cache“);
}
```

(b)

Figure 3.17: Example SEFF as FSM and corresponding source code

Example 3.2 (FSM-SEFF). In figure 3.17(a), a SEFF is modeled as a FSM for the provided service `read`, whose source code is shown in Figure 3.17(b)). This service first initialises a file handle, writes to a log file, and then reads from a cache within a loop. After completing the file access, another entry is added to the log. In the FSM, edges represent calls to required services and are annotated with the name of these services. The states abstractly represent the internal computations of a service after or before executing a required service. Notice, that the SEFF only contains the sequence of calls to the required services, while the component internal activity of initialising the file handle is abstracted.

Although SEFFs reveal the inner dependencies between provided and required interfaces of a component, they do not violate the *black box principle*. First, these specifications are only used by tools performing analyses, and do not have to be understood by humans. Second, they do not reveal the intellectual property of component developers encoded in the service's algorithms, because they are a strong abstraction of the component's source code. Third, in many cases, these specification can be generated out of byte code components, which are generally considered black box components.

SEFFs can be specified for *basic components* by the component developers and computed for *composite components* out of the SEFFs specified for the inner components [12]. For existing legacy *basic components* with available source code, the SEFFs have to be specified manually so far. However, in the future it is planned to implement analysis tools for component source code to assist component developers in the SEFF specification of legacy components by semi-automatically generating them.

3.1.4.4 Alternative types of Service Effect Specifications

Different *types* of SEFFs besides simple service lists and FSMs can be modelled to support different kinds of analysis (e.g., protocol checking, QoS analysis, etc.). If different SEFF types are defined for the same provided service, a mapping should exist between the FSM SEFF and the other types of SEFFs, which ensures the same names for provided and required services and the same order of calls to required interfaces.

SEFFs have been introduced by Reussner [12], who has used them in the context of parameterised contracts for *protocol adaptation* (Section 2.3). In that work, counter-constraint automata are used to model SEFFs restricting the number of calls to specific required services. Furthermore, using Petri nets to model SEFFs is envisioned to support concurrent component behaviour. However, assuring protocol interoperability is not possible if the component behaviour is modeled with Petri nets, because the language inclusion problem is undecidable for them in the general case.

While plain FSMs are well-suited for restricted protocol checking, they are generally insufficient for *QoS analyses*, because additional stochastic information and QoS characteristics (such as execution times or reliability values) are needed. Thus, several other forms of SEFFs have been proposed. For reliability prediction, Reussner and Schmidt [20] enhance SEFF FSMs with transition probabilities, so that they become Markov models. Similar Markov models enhanced with distribution functions for execution times have been used for performance predictions by Firus et. al. [4]. Happe et. al. [28] propose modeling SEFFs as stochastic Petri nets to enable QoS analyses involving concurrency. Koziolek et. al. [27] use stochastic regular expressions as SEFFs to make component-based performance predictions. These expressions are similar to Markov models, but are hierarchically structured and contain special constructs to model loops. Koziolek et. al. [3] use annotated UML 2.0 activities as SEFF models in the context of performance analysis. In [1] so-called *resource demanding SEFFs* have been introduced for QoS analysis, which have become part of the PCM and are described in Section 3.1.4.2.

In the PCM, a *basic component* can contain any number of SEFFs for each provided service, but at most one SEFF of each type, such as FSM or Petri net. A restriction on a particular SEFF type is deliberately avoided to enable different kinds of analyses. At the point of writing, the only SEFF type explicitly included in the PCM is the *resource demanding SEFF*. However, other types can be included in the PCM by inheriting from the class `ServiceEffectSpecification`. Consistency between different SEFF types has to be ensured by component developers, as it is not checked by the component model. If component developers implement a component based on a SEFF, it has to be ensured that the language of the SEFF is a superset of the language of the implementation.

3.1.5 Parametric Dependencies

Parameters of component services may have a significant impact on the perceived performance and reliability of a component-based systems. A major problem for component developers 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, component developers cannot specify fixed values. However, to help the software architects in QoS predictions, the component developer can specify the *dependencies* between input parameters and resource demands, branch probabilities, or loop iteration numbers in SEFFs. If an usage model of the component has been specified by business 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.

It can be distinguished between

- **Input Parameters:** which are passed to a component service by its clients (users or other components). Possible input parameters are specified in the signature of the interface described by a SEFF.
- **Output Parameters:** return values, which are sent back to clients by a service after finishing its execution. Possible output parameters are specified as return parameters in the signature of the interface described by a SEFF.
- **Configuration Parameters:** which can be global variables or configuration options of a component. Component developers specify configuration parameters for a component, and can refer to them in all SEFFs of that component.

All of these forms of parameters can cause different influences on the QoS properties of a system:

- **Resource Usage:** Parameters can influence the usage of the resources present in the system executing the component. For example, the time for the execution of a service that allows uploading files to a server depends on the size of the files that are passed as input parameters. In this case, the parameter alters the usage of the storage device. Another example would be a service for sorting items within an array. The duration of executing the sort service would mainly depend on the size of the array passed to it. Thus, the parameter would alter CPU usage.
- **Control Flow:** SEFFs (see Section 3.1.4) describe how requests to provided services are propagated to other components. The transition probabilities or number of loop iterations in SEFFs can depend on the parameters passed to a service. For example, a component service might provide clients access to a number of databases, thus communicating with several database interfaces as required services. This service would call different required services depending on the input parameter passed to it. Thus, the transition probabilities in the SEFF modelling the alternative to communicate with different databases would directly be linked to the input parameter. Another example could be a component service having a collection parameter, which would call another component's service subsequently for each item in the array. Such a situation would be expressed as a loop in a SEFF, and the number of iterations would directly be linked to the size of the array.
- **Internal State:** Input parameters can influence the internal state of the component. The component state in turn may influence resource usage or control flow between components. Imagine a component allowing users to log in to a system, which stores user sessions as global variables. The later behaviour of other services of this component in terms of control flow propagation and resource usage could depend on which user is currently logged in. Thus the QoS properties of the component would be related to the internal parameter, which was created when the user logged in to the system. Although the influence of the internal state has been recognised by us, it is so far not modelled in the PCM and remains future work.

3.1.5.1 Using Parametrisation in an RDSEFF

Actions in an RDSEFF can be parametrised in dependency on the RDSEFFs input parameters, on output parameters of called services, or on configuration parameters of the component containing the RDSEFF. For configuration parameters, component developers can specify default values.

The following actions in an RDSEFF can be parametrised:

Resource Demands Resource demands of a component service may vary depending on how the service is used. For example, the hard disk demand of a component service, which offers downloading different files from a server, strongly depends on the size of the file that is requested via a parameter. Another example would be the CPU demand of a component that allows sorting collections. Its CPU demand for the sort operation would depend on the number of elements in the collection. Thus, it could not be specified as a fixed value by the component developers, because they cannot forsee how the component will eventually be used by third parties. Therefore, it is necessary to specify resource demands in dependency of parameters.

To do so, the `ParametericResourceDemands` can also contain a demand in dependency to the SEFF's parameters (e.g., demand="x.BYTESIZE * 200", where "x" is an input parameter of the service). Once "x.BYTESIZE" is specified by third party users, the actual resource demand can be computed.

Control Flow The control flow may also vary depending on how the service is used. For example, the number of order items might determine how often a loop to check the item's availability is executed.

For branches, `GuardedBranchTransitions` contain a branch condition as a random variable. For example, a branch condition could be connected to the value of an parameter ("x.VALUE $<$ 1"), in which case a branching probability could be computed once the value of the parameter is known (cf. Section 3.1.5.1).

Both types of loops (`LoopActions` and `CollectionIteratorActions`) can also depend on parameters. For a `LoopActions`, the number of iterations is a random variable, and this random variable can include dependencies on parameters. For example, the number of loop iterations might be specified as ("5+x.VALUE"). Note that x.VALUE must have integer values in this case, because fractions of loop iterations are undefined.

`CollectionIteratorActions` enables modelling the special but common case of iterating over a collection. Because of this, `CollectionIteratorActions` have to reference an parameter of the current component service. This parameter must be a collection parameter and the number of elements in this collection has to be characterised with a random variable. Then the loop gets executed for each element in the collection.

Notice, that for `LoopActions`, it is assumed that the parameters characterisations used in the loop body are *stochastically independent*, whereas for `CollectionIteratorActions` it is assumed that the characterisations are not independent. For example, if the characterisation of a parameter value is specified by a random variable and is used by two external call actions within a loop body, the analyses algorithms have to assure, that the second action uses the same characterisation as the first action and that the random variable does not get evaluated a second time.

External Calls It is possible that input parameters passed to a required service do not receive fixed or constant values within the calling component service. They might in turn depend on parameters of the calling service. These parameters are however unknown to the component developers. Therefore, in such a case, the component developers have to specify a *dependency* (instead of a constant characterisation) between parameters of the calling service and input parameter passed to required services.

In the PCM, VariableUsages can be used to specify the needed dependencies between parameters (Figure 3.11), which abstractly characterise the data flow through a component service. These variable usages are aligned to the parameter model described in Section 3.1.5.2. With them, it is possible to characterise the value, byte size, or type of primitive parameters as well as the number of elements or the structure of collections. The characterisations can be expressed as random variables (for details refer to Section 3.1.5.2).

Example In the following, examples for the specification of parametric dependencies in the PCM will be illustrated. Note that as a concrete syntax a more easily readible UML-based notation is used for the examples instead of the abstract syntax.

As the first example (Figure 3.18), 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 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.

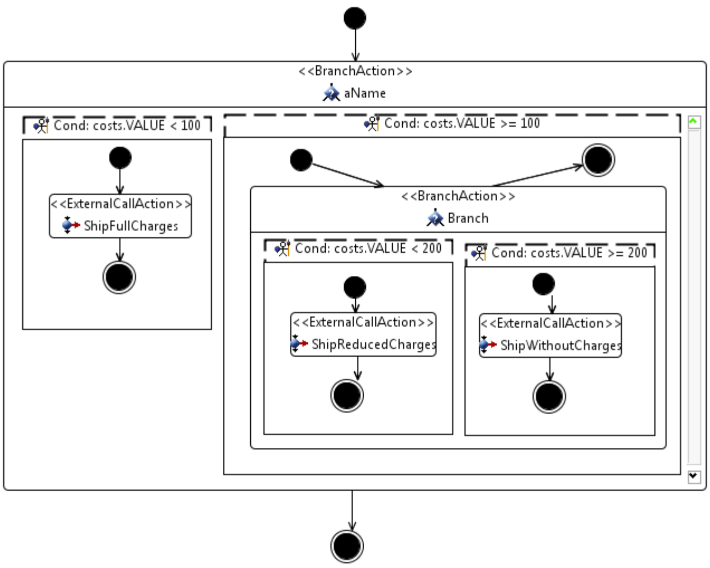


Figure 3.18: Branch Condition Example

Once a domain expert specifies the value of the parameter costs, it can be derived which of the services will be called.

The second example (Figure 3.19) illustrates assigning a number of iterations to a loop in a parameterisable way. The illustration 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.

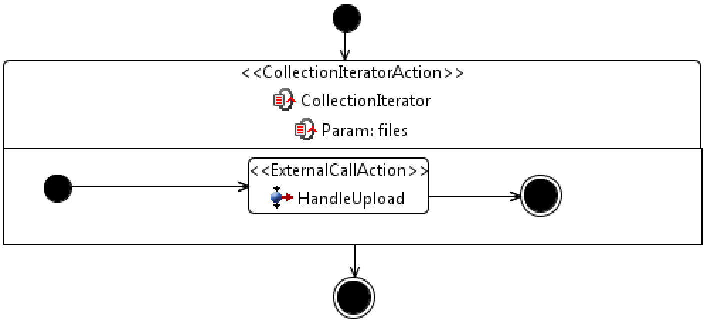


Figure 3.19: Loop Example

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 once the number of elements are known. 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.

3.1.5.2 Structure

The PCM parameter model (Figure 3.20) allows characterising actual parameters of a component service by associating a `VariableUsage` with a formal `Parameter`. The formal `Parameter` is part of an interface from the repository model (see Section 3.1.2) and referenced from the parameter model using an `AbstractNamedReference`. This may be a `NamespaceReference` or a concrete `VariableReference`, which contains the name of the parameter to be characterised. With `NamespaceReferences` more complex data structures such as composite data types or the inner elements of collections can be referenced. For example, an object 'customer' containing two strings 'name' and 'address', can be characterised by providing characterisations for both 'customer.name' and 'customer.address'.

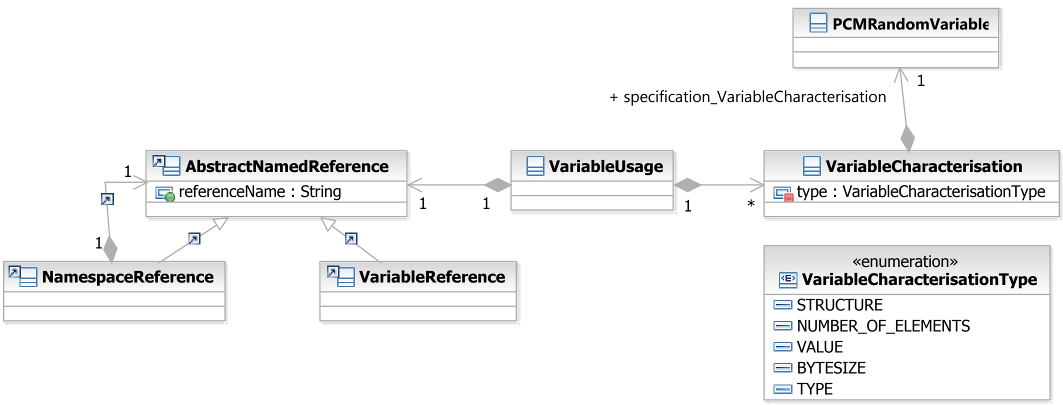


Figure 3.20: Parameter Model

Note, that it is only necessary to characterise parameters if they indeed influence performance or reliability. Many parameters do *not* change resource usage or alter the control flow between components, and their characterisation can be omitted. Characterising every parameter of the services in a complex component-based architecture would require too much effort and not support performance analysis.

Many parameters can be characterised by simply providing a constant value for them. However, as motivated in the example above, in some situations it is useful to characterise parameters not only with constant values but with *probability distributions* to allow more fine-grained predictions. Thus the attributes of parameters are characterised with `VariableCharacterisation` in the PCM, which inherit from `RandomVariables` (see Section 2.5).

Different attributes of parameters can be characterised in the PCM parameter model. Primitive data types can be characterised with their value, byte size, or data type. To demonstrate the modelling capabilities, consider the examples for primitive parameters in Figure 3.21. Note, that these examples are illustrated with class diagrams instead of the annotated activities used before.

Example 3.3. In Figure 3.21(a), a probability distribution for the *value* of the integer parameter named "id" has been specified. The parameter receives the values 1, 2, or 3 with probabilities of 70%, 20%, and 10% respectively. Figure 3.21(b) shows a parameter named "inputFile", whose *size in bytes* has be specified as a constant value (20). Via inheritance, extensions of certain parameters may be passed to a component service, thus the concrete *data type* may additionally be characterised by a domain experts. In the example in Figure 3.21(c), the parameter "shape" of type GraphicObject may become a circle, triangle, or rectangle, which may alter the response time of the service that is supposed to draw these graphics. It is also possible to specify multiple characterisations of a single parameters, for example to specify the value *and* byte size.

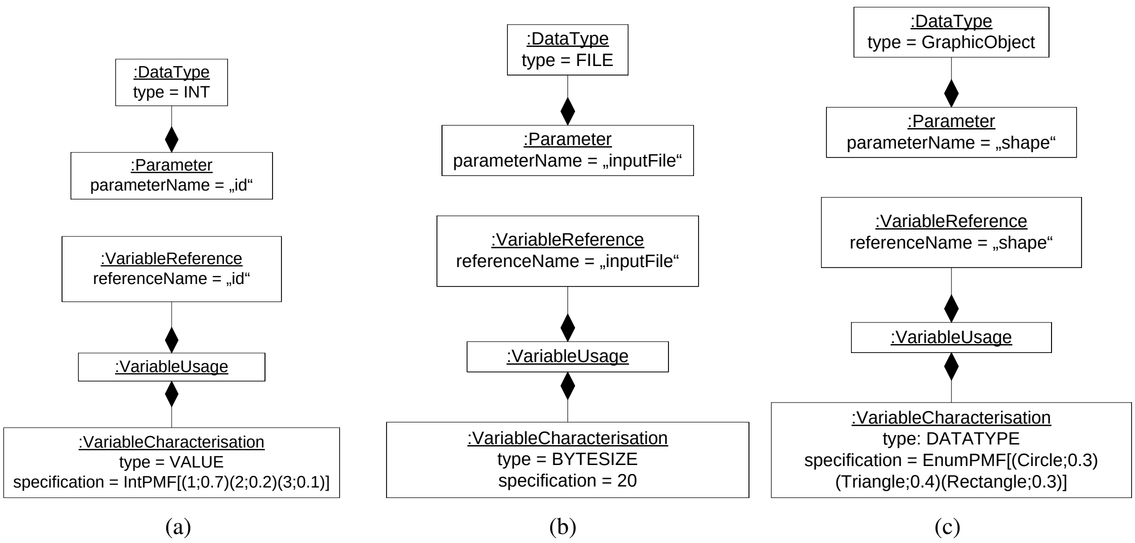


Figure 3.21: Primitive Parameter Characterisation Examples

Example 3.4. For *collection parameters*, it is more difficult to characterise the value domain. The performance-influence of collections like array, tree, or hash can sometimes be characterised simply by the *number of elements*. Thus, it may be appropriate for such parameters to specify probability distributions over the number of elements. Consider the example in Figure 3.22(a): the number of elements in the collection “niceTree” of type RedBlackTree has been specified with a probability distribution, i.e., the tree contains 10, 100, or 1000 nodes with probabilities of 10%, 30%, and 60%. The value, byte size or data type of a collection can be characterised as explained above. In Figure 3.22(a), the size of the collection has additionally been specified.

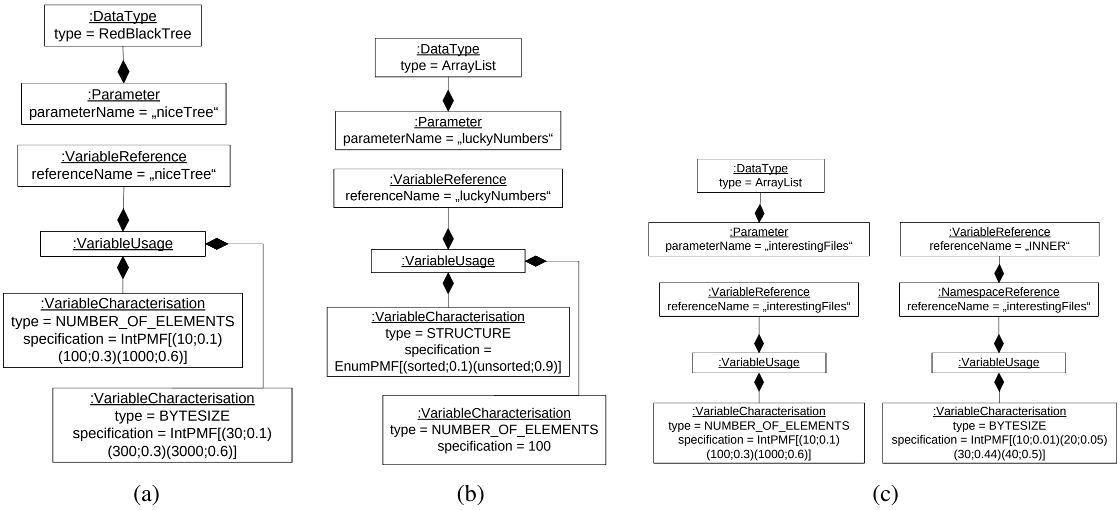


Figure 3.22: Collection Parameter Characterisation Examples

Besides the number of elements, it is sometimes useful to specify the *structure* of a collection, if it influences QoS properties of a component service. For example presorted arrays may by sorted quicker than unsorted arrays or the deletion duration of an element in a tree may depend on the balance of the tree. In Figure 3.22(b), the structure of the array list “luckyNumbers” has been characterised as sorted with a probability of 10% and unsorted with a probability of 90%. Additionally, the number of elements in the array list has been characterised with the constant value of 10.

To ease modelling, collection contain may contain one inner `VariableUsage`, which shall representatively model the *inner elements* of a collection. In the example in Figure 3.22(c), the collection “interestingFiles” is characterised with its number of elements. Additionally, the inner parameter usage representatively characterises a single file within the collection. Here, the files in the collection have a size in bytes between 10 and 40 bytes.

3.1.5.3 Related Work

Many performance prediction approaches or performance related meta models neglect the influence of parameters values to the above described properties. The UML SPT profile [29] as well as the CSM [30] do not include notions of parameters. Methods that build on these modelling approaches such as CB-SPE [31] thus also cannot express the influence of parameters to QoS properties.

KLAPER [32] allows characterising parameters values, but does not include a formal way of creating abstractions for parameters, because the kind of parameter specification is left open for developers. This limits the use of tools evaluating KLAPER instances, because they can not foresee all possible ways of abstracting parameters. Thus, manual work is required with KLAPER to complete the performance prediction process if parameters are involved.

The ROBOCOP component model [33] also allows characterising parameter values. However, as

ROBOCOP aims at embedded systems, it is assumed that the domain for parameter values is very limited. It is possible to model parameters with constant values only, stochastical characterisation for parameter abstractions are not in the scope of that work.

The performance prediction approach by Hamlet et. al. [26] models components as functions and divides their input space into several subdomains. For each subdomain, which can be conceived as a parameter abstraction, different execution times can be determined. However, subdomains are always built for the values of parameters in this approach, other attributes of parameters are neglected.

3.2 Software Architect

3.2.1 Overview

The tasks of the software architect are to retrieve components from existing repositories and connect them to build an assembly which is an essential part of the complete system. Connections are specified by using system assembly connectors to connect required roles of components with provided roles of other components. After connecting all components, the software architect puts the components into a system and defines the system provided and system required roles as well as the respective delegation connectors. The definition of a system and its boundaries is comparable to the definition of composite components. However, the difference is that composite components are built with the aim to use them in other composite components or assemblies. On contrary, systems are built to interact with other systems only. An overview of a system and its subconcepts is shown in figure 3.23.

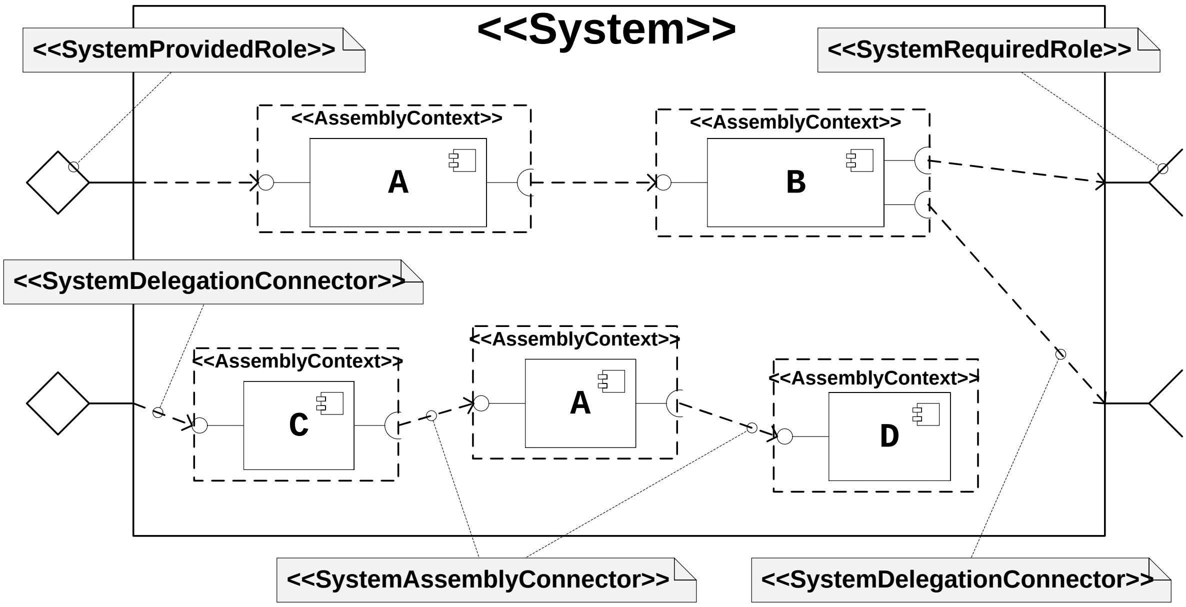


Figure 3.23: A system and its assembly

Components can only be used in contexts as introduced in section 2.4. Hence, the software architect is responsible to introduce system assembly contexts in which a component is put. When a component is put into a context its roles also become part of the context. Such roles which are part of a context can be connected. For this, a required role in a specific context is connected to a compatible provided role in an other context. A single component can be put into several different contexts and can be connected differently in each of them. As mentioned in section 2.4, the introduction of multiple assembly contexts is important as they capture the different influence of component external calls in different contexts.

The defined assembly model is finally passed on the the system deployer who specifies the allocation of the components to middle- and hardware environments. The assembly model is the second essential part of a system and is described in detail in section 3.3.

3.2.2 Assembly

An assembly is a set of assembly contexts containing component types from several repositories and a set of system assembly connectors connecting the components in their context. Conceptually, every system has exactly one assembly. An assembly is different from a composite component in its visibility for the system deployer. The inner structure of a composite component is hidden from the system deployer. Only the outer aspects of the component are visible for the system deployer which is mainly the component and its roles. Opposed to this, the system deployer has full access to the assembly contexts and the system assembly connectors. The rationale behind this difference in modelling is that a composite component should always look like any other component (besides for the developer of that component). The decision, whether a component's inner structure is build from scratch (i.e., as basic component) or by connecting existing components (i.e., using a composite component), is considered as an implementation detail. As a consequence the inner structure of any component is only visible to the component developer. Neither the assembler nor the deployer know about the inner structure. To be consequent this means that a composite component *can not* be allocated on *more than a single runtime environment* as this would mean that the system deployer has access to the composite components inner structure. This is different for an assembly. The component and their contexts as well as the system assembly connectors are visible and can be distributed in arbitrary ways by the system deployer on execution environments.

3.2.3 Assembly Context

As introduced above, the software architect uses assembly contexts to put components into a component assembly. Contexts support the multiple use of the same component type in several environments in an assembly.

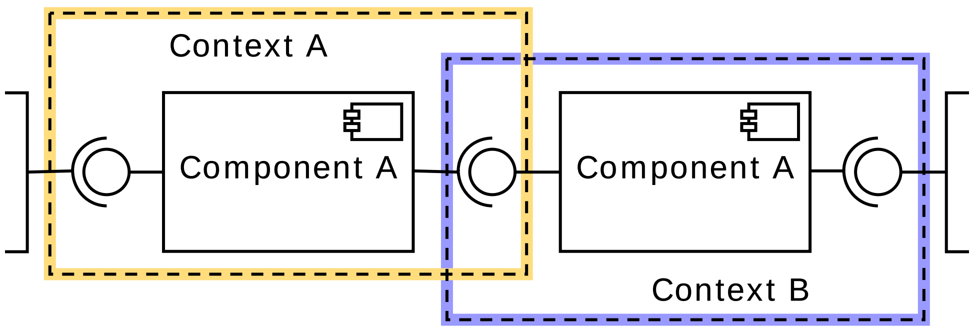


Figure 3.24: Component Assembly Context

The assembly context refers to exactly one component from an arbitrary available repository for which the context is applied. The component and its provided and required roles are affected by the context in which it is used. This can be indicated by deriving from the provided and required roles the corresponding provided and required context roles.

According to the principles of parametric contracts (see section 2.3, context roles represent the contextual influenced interfaces of the component in a given assembly context.

At the level of assembly contexts, software architects may also set configuration parameters of the included component and thus override the default values specified by the component developer.

3.2.4 System Assembly Connectors

After putting components into assembly contexts (from which provided and required context roles can be derived) they can be connected by using system assembly connectors. A system assembly connector connects a required role in of a component in a given assembly context with the provided role of a component in a different assembly context [2]. The meaning of the connector is that any call of the client component using the required role involved will be routed to the provided role of the server component.

Connectors are important entities when it comes to checking of interoperability classes. The minimum requirement a connector has to fulfil is that the interface of the provided role is a supertype of the required role. This automatically implies semantic conformance of the interfaces (compare section 3.1.2).

3.2.5 System

As mentioned in the overview, an assembly forms on of the important aspects of a system. A system consists of an assembly and an allocation as described in section 3.3.5. The first specifies how the components are connected with other components, the latter specifies how the components and connectors are mapped to hardware and middleware environments. Systems can be seen as special kind of composite components - with the visibility differences mentioned above and the fact that an allocation is also provided. Systems are not supposed to be reused as components are. The are assumed to be coupled by using special techniques for system integration.

3.2.6 Subsystem

A system may contain subsystems. A subsystem shares the same conceptional elements of a composite component, and thus contains encapsulated components. Compared to composite components, the encapsulated components of a subsystem can be allocated on different hardware or middleware resources. A composite component and all its encapsulated components can only be allocated on the same hardware or middleware resources.

3.2.7 System Roles

As components, also systems can specify that the offer the functionality of a specific interface or that they require functionality of a specific interface. Analogous to the component roles, the PCM defines system provided and system required roles. The semantics corresponds to the semantics of the roles of a complete component type. The system offers the functionality specified in the provided interfaces if all requirements of the system are met. If they are not met, only a subset will be offered. The semantics of the required interfaces is that a system may call other systems using a required role. It can not call other services than those defined in the system required roles. Using parametric contracts (see section 2.3) for functional dependencies, the actuall demand or the actual provided functionally can be derived (which would result in a system context role, but as it can be fully derived, it is not part of the PCM specifications).

[2]Using the derived context roles as concept, we can say, a system assembly connector connects a required context role and a provided context role

3.2.8 System Delegation Connectors

Systems can have delegation connectors, just like composite components. The delegation connectors are used to route calls to the system interfaces to the desired destination. As composite component delegation connectors there are also two types of system delegation connectors: provided and required. Provided system delegation connectors route calls to system interfaces to components in the assembly which are responsible to process the requests. System required delegation connectors route calls of components in the assembly, which are not processed in the current system, to system required roles. Hence, they can be used to model calls to other systems.

3.3 System Deployer

3.3.1 Motivation

To execute an application specified by a component assembly, components and connectors have to be allocated to different hardware and software resources, which provide the required infrastructure. Servers, clients, or any other kind of systems are set up with the required operating system and middleware. Components are installed on the systems and configured so that they can run in this environment. Computers are connected by networks enabling the communication needed by the components. The whole process of setting up the infrastructure, allocating components, and configuring the system is handled by the deployer as introduced in section 2.1.

For QoS analyses, it is required that the deployment of the software architecture is specified in advance, since it has a major influence on QoS attributes, such as performance and reliability. For example, the response times of a component's services will be shorter when it is deployed on a machine with a 3GHz processor instead of a machine with a 1GHz processor. With the specification of the execution environment with its hardware and software resources and connections, and the component allocation, several QoS attributes can be predicted. So, deployers are able to try different deployment scenarios to find the optimal configuration for a software architecture. In many cases, such a procedure can save a lot of work and cost, since bottlenecks can be discovered early and hardware will not be oversized.

In the context of the PCM, we currently provide a basic model for the description of resource environments and allocation of components. In the following, we describe how these concepts can be used to specify new resource types that form an execution environment. Furthermore, we introduce allocation contexts that allow us to allocate components to multiple hard and software nodes. For the future, it is most likely that the model described here will be extended to allow a higher accuracy in terms of modelling as well as prediction results.

Section 3.3.2 describes the responsibilities and duties of deployers. In section 3.3.3, we describe what kinds of resource types we model and how they can be used. Section 3.3.4 shows how the PCM in combination with a fixed set of resource types can model an execution environment. In section 3.3.5, we describe how components are allocated on resource containers and how they can access the available resources. Finally, section 3.3.6 sums up open issues and assumptions of our model.

3.3.2 Responsibilities of the Deployer

Mainly, the resource environment is in the deployer's responsibility. This includes the specification of resource environments as well as the installation of a component-based application or the setup of a new environment. Deployers are assumed to be experts in the area of component deployment (allocating software components to different hard- and software nodes) and the configuration or creation of an environment, that enables the system to fulfil its extra-functional requirements. Deployers are responsible for:

- Definition and description of the resource environment. This includes the specification of hard- and software resources, their properties, and their interconnections.
- Allocation of components to different resources.

Deployers are not only concerned with the specification of the resource environment and component allocation, but also with the realisation of the actual system setup. However, as these are two different tasks, they might not be performed by a single person only. For example, an application for the mass market might have a set of typical deployment scenarios defined by members of the development team, but the setup will be accomplished by the customers themselves.

To specify the resource environment in the PCM, deployers use *resource containers*. A resource container represents a part of the real world that can host components, for example an application server or client computer. It holds a set of different resources, such as processors, hard disks, or thread pools. Each resource conforms to a certain *resource type* that discribes a class of resources with common properties. If a component is allocated on a resource container, it has access to all resources the container provides.

3.3.3 Resource Types

A *resource type* describes the common properties of a class of resources. For example, a *processor* type could be used to describe different CPUs, e.g. with a different clock speed or a different architecture. The concept of resource types allows a flexible specification of different kinds of resources that might occur in a real world scenario. Component developers and deployers agree on a common set of resource types that is specified within a so-called *resource repository*.

We distinguish *passive* and *processing resources*. Passive resources can only be owned by a process or thread, while processing resources do some work by themselves and offer processing services. A scheduler might decide, which process is handled next by the processing resource. CPUs and hard disks are typical processing resources, while connections to a database or a block of memory are passive resources. *Communication links* are a special kind of processing resource type used to describe connections between different resource containers.

Resource pools manage a limited set of resources of the same type. Typical examples are database connection pools and thread pools. A process or thread can fetch one database connection, use it to read or update some of the database entries and then return it to the pool. If no database connection is available, the process will block until one is available in the pool.

Semaphores are the most basic kind of passive resources. They can be used for synchronisation and limiting access to a resource. Basically, a semaphore is an integer value with an acquire (or `p`) and release (or `v`) operation. Intuitively, the value of a semaphore indicates how many instances of a resource are available. If the semaphore is greater than zero, the acquire operation reduces the semaphore counter by one and continues the execution. Otherwise, it waits until the counter is greater than zero. The testing and setting of a semaphore's value has to be atomic (i.e., it must not be interrupted). The release operation increases the counter by one, which must be atomic as well, and awakes the waiting threads or processes.

Acquire and release actions are used for semaphores as well as for resource pools and can be directly modelled in the service effect specification (see section 3.1.4).

For communication resources, we consider any kind of network connection. The rate or throughput of a connection is specified in megabytes per second. This resource type can be used to model most of the common networks. For example, a wireless connection between two nodes can be described as an ethernet connection with a throughput of 11 MB/s.

The resource types described here can be considered as a basic set, which has to be extended and refined in future. Next, we describe how these resource types can be used to specify an execution environment.

3.3.4 Resource Environment

In the PCM, resource environments are described by a set of resource containers and connections between them. A resource container provides a set of processing and passive resources to the components it hosts.

It represents a physical entity such as a server, a desktop computer or an element on a higher level like application servers or web browsers.

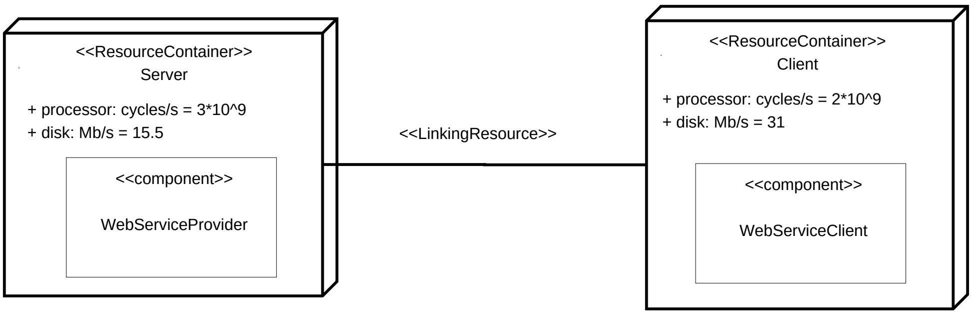


Figure 3.25: Simplified example of a resource environment.

Example 3.5 (Resource Environment). Figure 3.25 shows a simplified view on a resource environment. The depicted system consists of two resource containers, a server and a client, and a linking resource between them. The figure also shows the allocation of two components, a `WebServiceProvider` and a `WebServiceClient`. Figure 3.25 is a structural view of the resource environment. For each container, a processor and a disk are specified. Both have different performance values for the resources they provide. For instance, the processor of the server has a clock frequency of 3 GHz, while the client has a clock frequency of 2 GHz.

A container also allows for specifying the container's operating system. If specified, the performance simulation makes use of more detailed scheduler model. In the example, the server resource container is specified to run Linux 2.6, whereas the client is specified to be a Windows XP machine. Specifying an operating system is not necessary. If omitted, a more abstract scheduler model is used.

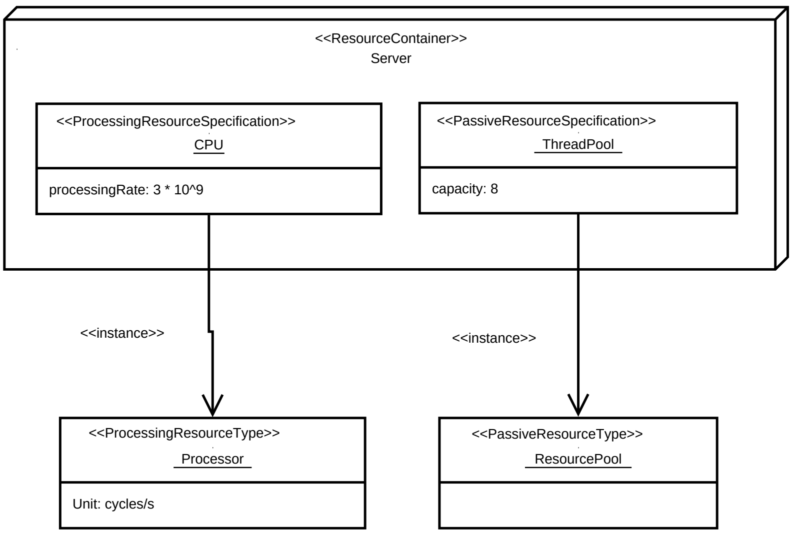


Figure 3.26: Specification of Resources of a Container.

Figure 3.26 shows the resource specification in more detail. The server contains a `CPU` and a `ThreadPool`. Both are described by `ProcessingResourceSpecifications`, which characterise the QoS relevant attributes of a resource and relate it to a resource type. There is a dual-core CPU with a processing rate of 3GHz and a thread pool that limits the degree of concurrency within the system. The CPU is a processing resource of the type `Processor`. The thread pool is a passive resource with a capacity of eight threads. The thread pool is of the type `ResourcePool`, which is depicted by an association to the type instance. For sake of simplicity, we omitted the modelling of any kind of data storage and hard disks at this point.

3.3.5 Allocation Context

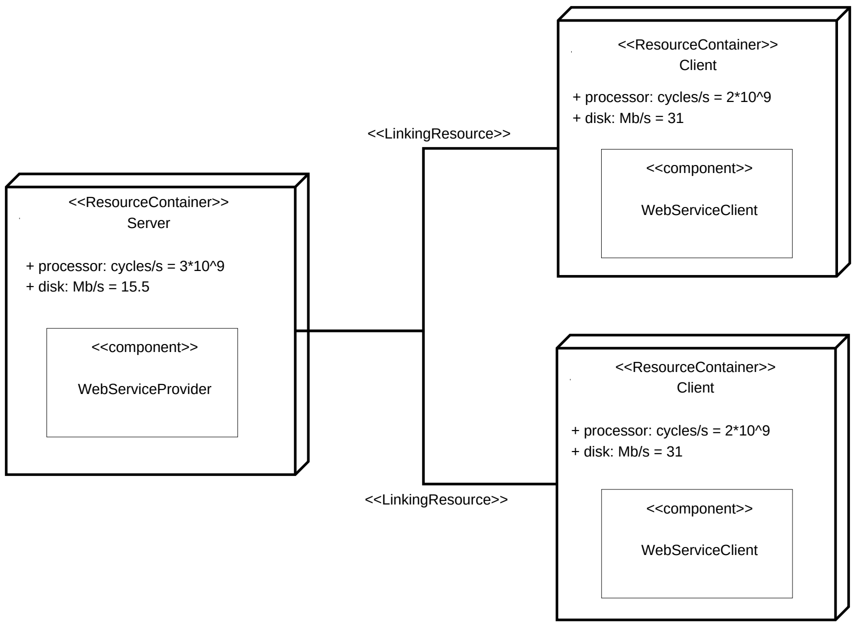


Figure 3.27: Alternative allocation for figure 3.25.

After introducing different resource types and means to specify execution environments, which provide the infrastructure to an application, components have to be allocated on the available resource containers. For this purpose, the PCM uses the *allocation context*. In section 3.2.3, we described how a component is integrated in a system assembly using assembly contexts. The idea of allocation contexts is similar. Each component integrated in an assembly might be allocated on multiple resource environments. Thus, for each component in an assembly context, there can be multiple allocation contexts that place the component on different resource containers. For example, a possible alternative of the allocation in figure 3.25 is shown in figure 3.27.

Figure 3.28 shows a simplified instance of the PCM that realises the allocation shown in figure 3.27. The allocation context is an association class that links a component to a resource environment. The allocation context allows to specify the placement of the same component on multiple resource environments. In reality, a copy of the component is created for each machine. Furthermore, the allocation context stores QoS related information that depends on the resources used by a component. For example, if an internal action of a component uses 5000 cycles on a `Processor` resource, this can be transformed to an execution time of $1.\overline{6}\mu s$ for a processor with 3GHz ($1/(3*10^9 s^{-1})*5000$). As the execution time of internal actions depends on the resources the component is allocated on, these information are handled by the allocation context.

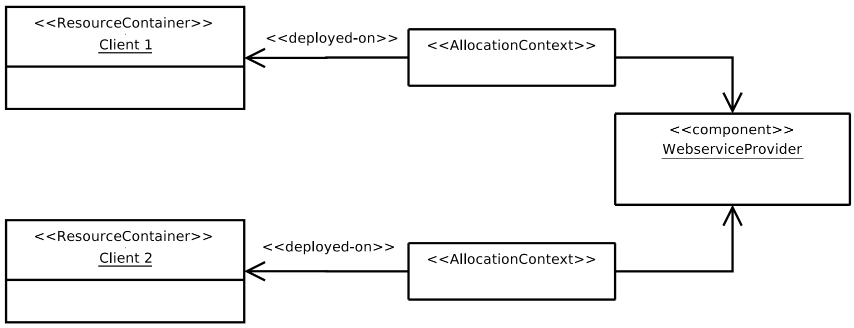


Figure 3.28: Allocation of a component on multiple resource environments (simplified).

At the level of allocation contexts, system deployers may also set configuration parameters of the included component and thus override the default values specified by the component developer or the values specified by the software architect.

In the PCM, resource environments are described using resource containers holding an arbitrary number of processing and passive resources. Linking resources connect resource containers with each other and provide a communication resource for sending data from one container to another. Resource types can be used to specify which kinds of passive, processing and communication resources exist. Components that are integrated into an assembly can be allocated on resource containers using allocation contexts. These allow to allocate one component on multiple resource containers and store QoS relevant information, which depends on the container, independent of the component. So, the PCM provides a complete infrastructure to specify the environment of an application and its allocation. However, there are a lot of open issues that need to be addressed in the future. We will discuss some of them in the following.

3.3.6 Open Issues and Future Work

So far, the PCM does not support the modelling of *hierarchical resource containers*. This is a major limitation for deployers, since they cannot model different software layers running on the same machine. For example, virtualisation of operating systems cannot be specified. Furthermore, it is not possible to describe systems that contain multiple components that are placed on different software layers, e.g. operating system and application server, but on the same machine.

Another limitation stems from the modelling of *linking resources*. At the moment, we only allow a single linking resource between two resource environments with one specification. Thus, scenarios in which two hardware nodes are connected by multiple links, e.g. LAN and wireless LAN connection cannot be modelled. Furthermore, it is not possible to explicitly allocate connectors between components to linking resources. With only one connection between two containers, this can be done automatically using direct links only. However, if multiple connections are allowed the allocation of connectors must be modelled explicitly. The same problem arises when indirect communication between containers is allowed. In this case, the communication path between components is ambiguous even with only one connection between two containers.

In section 3.3.3, we described how to specify new resource types. Even though this provides a high flexibility, it requires component developers and deployers to agree on a common set of resource types. For scientific purposes, this is feasible. However, we need to integrate a *standardised set of resources* into our model so that there are no mismatches between the specifications of different parties. As the modelling of execution environments is not as elaborated as other parts of the PCM, we left the specification of resources open for the time being. For the future, we plan to fix the available set of resources.

Also, the *specifiable properties of the resource types are limited*. So, if a new resource type has additional attributes that have to be specified, this cannot be described. For example, queues could be introduced as a passive resource. Usually, queues are used for asynchronous communication between multiple processes and threads. One process puts a message or any data into the queue while another process reads it. A producer-consumer system is a common example for such a scenario. A special application for queues can be found in combination with active objects [34]. Instead of calling a method of an active object directly, the call with its parameters is placed in a message queue. The scheduler of the active object fetches the messages from the queue and processes them. Queues do not only have a capacity as all passive resources do, but also require an attribute which specifies the order in which its items are processed, like LIFO or FIFO. This is not possible so far.

3.4 Domain Expert

3.4.1 Overview

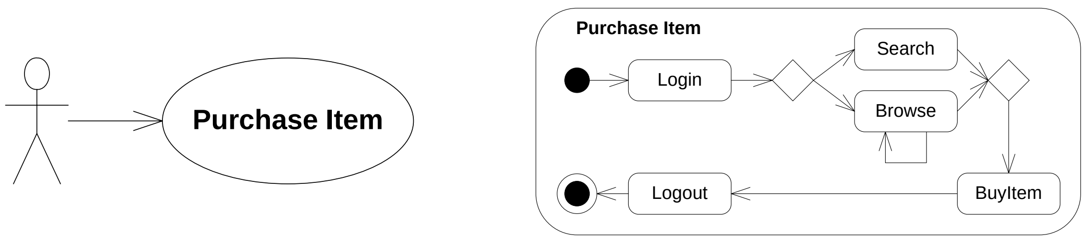
Business *domain experts* participate in the development of any larger software system. This role has special knowledge and experience in the business domain (e.g., automobile, banking, etc.) of the system being developed. However, domain experts usually have no or only a limited technical background. They mainly participate in the development process during feasibility studies and requirements analyses and help in specifying the functionality and business logic of the system. Therefore, they have to interact closely with the system architects, who have a technical background and are able to tailor their requirements to a component-based software architecture.

For early *QoS analyses*, domain experts assist system architects in specifying the user interaction with the system. As they are familiar with the business domain and the targeted end-users, they should best be able to specify the anticipated usage scenarios and workloads of the system. The usage specifications may be based on experiences with similar legacy systems or on market analyses of the business domain. In the PCM, the usage specification consists of usage models (see Section 3.4.2), which are similar to UML use cases with attached UML activities. They additionally contain stochastical information (e.g., probabilities of choosing a branch in an alternative) and the notion of workload to characterise the number of users in the system, which is especially relevant for performance predictions. Usage models may be refined with a parameter model (explained in Section 3.4.2.3) to characterise the data values passed to component services by users.

3.4.2 Usage Model

An instance of the PCM usage model specifies user interactions with a system. It describes which services are directly invoked by users in specific use cases and models the possible sequences of calling them.

3.4.2.1 Example



(a) Example UML Use Case (b) Example UML Activity

Figure 3.29: Example UML diagrams for using an Online Shop

Example 3.6 (Usage Model). For a first overview, Figure 3.29 shows a UML use case diagram and corresponding UML acitivity for using an online shop. Users log in to the shop, either search or browse in the shop's catalog, then buy items, and finally log out. Figure 3.30 shows the corresponding PCM usage model instance of this scenario. In this example figure, the concepts were illustrated with UML activities, where the stereotypes (denoted by «stereotype») refer to classes in the PCM. This simple usage model instance serves as a running example for the rest of this section.

Figure 3.30: Example Usage Model for an Online Shop

An usage scenario consists of i) a workload to model the frequency of user interactions and ii) a scenario behaviour to model the steps of service invocations by users. In this example, the workload is a *closed* workload (upper right corner of Figure 3.30) and specifies that 100 users (population) execute the scenario behaviour. Each user executes the actions specified in the behaviour from the start action to the end action. After reaching the end action, the user reenters the behaviour at the start action after 5 seconds (think time). The number of users in the system is fixed to 100 in this scenario.

The actions inside the behaviour either model flow constructs (start, stop, branch, loop) or user invocations of services available in system provided roles (Login, Search, Browse, BuyItem, Logout) (also see Section 3.2.7). Like in the UML diagrams before, users first log in to the online shop and then either search directly for an item via a search interface or browse the shop's catalog to find an item to buy. This alternative is modelled with a branch action and the probabilities of search and browsing are specified as 40% and 60% respectively. Browsing the catalog is modelled as a loop with three iterations, as it is assumed that users need three clicks to find the item they want to buy. After browsing or searching is finished, the user continues with buying the selected item, and finally the logging out from the shop.

Note, that usage models are completely decoupled from the inner contents of a system (see Section 3.2.5), which consists of an assembly (see Section 3.2.2) and a connected resource environment (see Section 3.3.4). The usage model only refers to services of system provided roles. It regards the component architecture (i.e., the assembly) as well as used resources (i.e., hardware devices such as CPUs and harddisks or software entities such as threads, semaphores) as hidden in the system. Thus, the usage model only captures information that is available to domain experts and can be changed by them. Resource environment and component architecture may be changed independently from the usage model by system architects, if the system provided roles remain unchanged.

3.4.2.2 Structure

The meta model for usage modelling in the PCM is described with more detail in the following (see Figure 3.31). A *usage model* consists of a number of usage scenarios. Each usage scenario is intended to model a use case of the system and the frequency of executing it. Thus, a usage scenario contains a `Workload` to model execution frequency and a `ScenarioBehaviour` to model a use case.

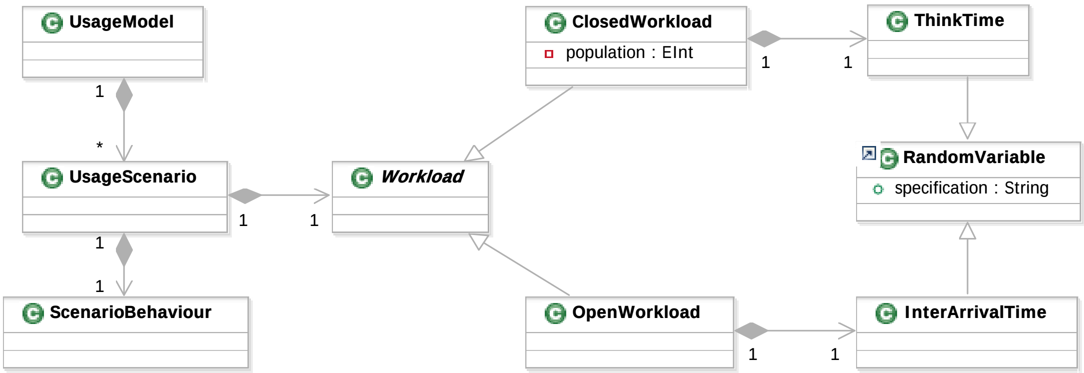


Figure 3.31: Usage Model: Scenario and Workload

Modelling *workloads* in the PCM is aligned with performance models such as queueing networks [35] or stochastic process algebras [36], the UML SPT profile [29] and MARTE [37]. Therefore, open and closed workloads can be specified. An open workload models an unbounded (thus open) number of users entering the system with a specific inter-arrival time as a random variable (e.g., a new customer arrives each 0.5 seconds) and leaving the system after executing their scenario. A closed workload models a bounded (thus closed) number of users entering the system, executing their scenario, and then re-entering the system after a given think time, which can be specified as a random variable (see Section 2.5).

Modelling *scenario behaviours* in the PCM (Figure 3.32) is similar to modelling resource demanding behaviours in SEFFs (see Section 3.1.4). However, SEFFs contain notions of resource usage, while the language for usage scenarios is reduced to concepts familiar to domain experts, and does not refer to resources.

`ScenarioBehaviours` contain a number of user *actions*. Within a scenario behaviour, the flow of actions can be described as follows: Each `AbstractUserActions` references at most one predecessor and one successor. `StartActions` initiate a scenario behaviour and contain only a successor, while `StopActions` end a scenario behaviour and contain only a predecessor. Notice, that the start and stop actions in the example above (Figure 3.30) follow this pattern.

Loops can be modelled to describe user actions that are repeated multiple times (e.g., searching for an item in a online store by repeatedly entering search terms, or repeatedly checking the latest status of an online auction). Loops over user service invocations are modelled with `LoopActions`, which are attributed with the number of iterations and contain inner `ScenarioBehaviours` to model loop bodies. These loop bodies may consist of multiple actions or even nested loops themselves. In the example (Figure 3.30), the browse action is called within a loop three times. It is additionally possible to specify the number of loop iterations with a probabilty distribution instead of a constant value to allow more fine-grained modelling (see Section 2.5).

Notice that the chain of user actions in a scenario behaviour must not contain cycles to model loops, i.e., an action referencing another action as its successor *and* predecessor. Instead, loops always have to

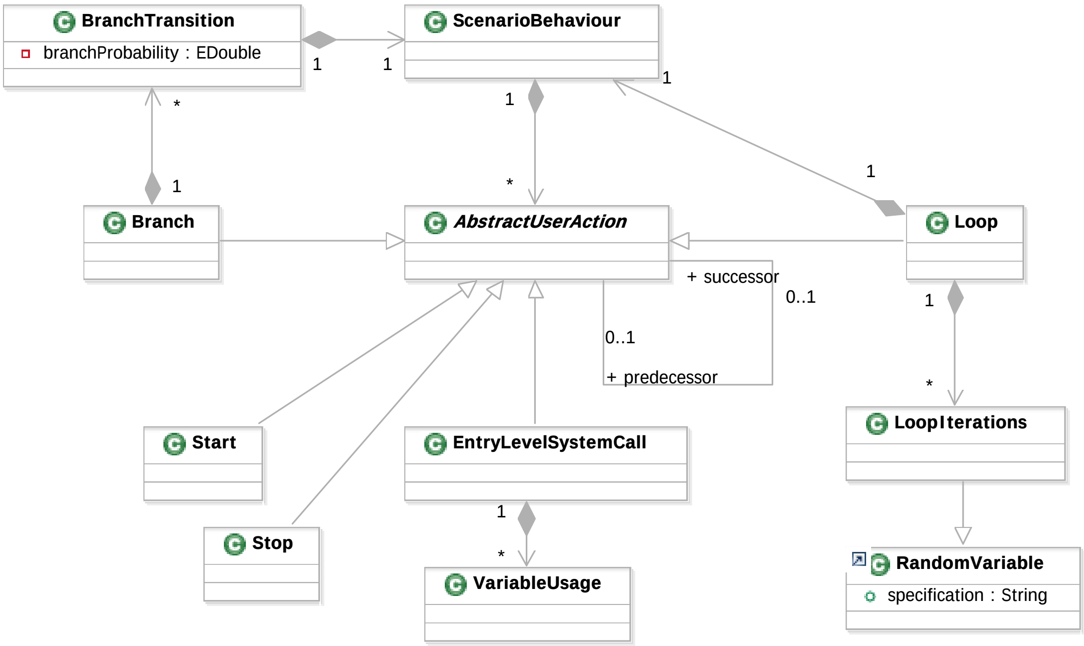


Figure 3.32: Usage Model: Scenario Behaviour

be modelled explicitly with loop actions. This explicit modelling eases the later analyses, as it arranges actions hierarchically in a tree structure, which can be analysed by standard tree traversal algorithms.

Most often, users have multiple choices to continue their interaction with the system. For such cases, the usage model offers *branch* actions, which are able to split the user control flow with an XOR-semantic and allow different successors to a single user action. A probability of executing each branch transition can be specified. In the example (Figure 3.30), users first log in to the system and then have the choice to either search the shop with a probability of 40% or browse in the shop's catalog with a probability of 60%. `BranchTransitions` contain inner `ScenarioBehaviours` to model the content of a branch. With this kind of modelling, additional merge actions for reconnecting two branches are not needed, as the control flow continues with the successor of the branch actions once the end action of the the branched behaviour is reached. Forks of user behaviour (i.e., splitting the flow with an AND-semantic) are not allowed so far, as it is assumed that a single user only executes the services of the same system subsequently but not concurrently.

Besides these control flow constructs, actual service invocations to the architecture are modelled by `EntryLevelSystemCalls`. They refer to services in system provided roles (see Section 3.2.7), which are connected to component services directly visible to the users. Inner component services, which are only called by other components cannot be referenced from the usage model.

3.4.2.3 Parametrisation

In addition to modelling the sequence of user actions, the domain experts also need to provide the values for performance-relevant parameters of the system. Component developers may have specified their component's service effect specification depending on external parameters (cf. Section 3.1.5). The domain experts need to specify the input parameter values that are used by users of the system (e.g.

the distribution of number of items bough in an online shop), and they need to specify domain-specific configuration parameters (e.g. the number of articles available in the database of this online shop).

Two short examples for specifying input parameters are shown in Figure 3.33. These examples extend certain actions from the usage model example in Figure 3.30 with parameter characterisations.

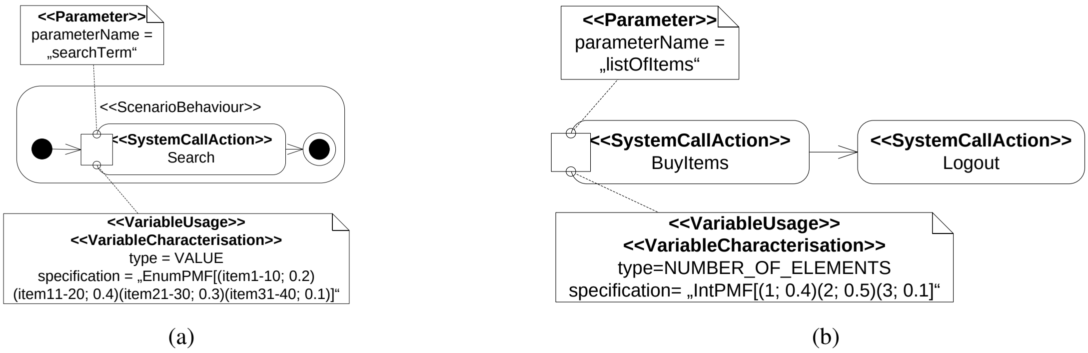


Figure 3.33: Parameter Characterisation Examples

Example 3.7. In Figure 3.33(a), a parameter 'searchTerm' has been introduced to the 'Search' action. The `Parameter` class of the PCM enables specifying a name and a data type (not shown here) for a parameter. Thus, it includes only information about the formal parameter. The actual parameter, i.e., the value a parameter takes when the service is actually called, can be characterised with a `VariableUsage`. In this case, the parameter is a string, which is the name of the item to be searched for. The database is assumed to contain 40 items. The domain expert has characterised the *value* of the input parameter and has specified a probability distribution for the search terms users pass to the service. Therefore, the domain expert has divided the input domain of the service into four subdomains (item1-10, item11-20, item21-30, item31-40) to reduce the modelling effort, and has provided probabilities for each of these subdomains. If the behaviour of the component service changes depending on which item is searched for (e.g., because of calling different databases), this can be included in the performance prediction, because the parameter has been characterised.

Example 3.8. In Figure 3.33(b), an array 'listOfItems' is passed to the 'BuyItems' action. The domain expert has not characterised the *value* of this array, but just the *number of elements* it contains. It is a suitable abstraction of the parameter in this case, because the value of the array is not relevant in this example. The service 'Buy Items' calls required services for each item in the array (not shown here because this is part of the service's SEFF and not the usage model), so that the number of elements in the array is sufficient for the performance predictions, as it is directly related to the number of loop iterations in the SEFF of this service. The number of elements is specified as a probability distribution, so that the loop is iterated with the same probability distribution.

Configuration parameters are specified in the usage model as `UserData`. They refer to an assembly context and contain a `VariableUsage` that overwrites configuration parameters with the same name that have been specified by component developers, software architects and system deployers.

3.4.2.4 Related Work

The PCM usage model has been designed based on meta models such as the performance domain model of the UML SPT profile [29], the Core Scenario Model (CSM) [30], KLAPER [32] and UML MARTE [37]. It is furthermore related to usage models used in statistical testing [38]. Although the concepts included in the PCM usage model are quite similar to the modelling capabilities of the UML SPT profile, there are some subtle differences:

- The usage model is aligned with the role of the domain expert, and uses only concepts known to this role. It is a domain specific language, whereas the UML SPT performance domain model is a general purpose language that includes information, which is usually spread over multiple developer roles such as the component assembler and the system deployer, so that a domain expert without a technical background could not specify an instance of it. Nevertheless, domain experts should be able to create PCM usage models with appropriate tools independently from other developer roles, because such models only contain concepts known to them.
- The number of loop iterations is not bound to a constant value, but can be specified as a random variable.
- The control flow constructs are arranged in a hierarchical fashion to enable easy analyses.
- Users are restricted to non-concurrent behaviour, as it is assumed, that users only execute the services of a system one at a time.
- System service invocations can be enhanced with characterisations of parameters values, as described in Section 3.4.2.3.

3.5 QoS Analyst

QoS analysts collect and integrate information from the other roles, extract QoS information from the requirements (e.g., maximal response times for use cases), and perform QoS analyses by using mathematical models or simulation. Furthermore, QoS analysts estimate missing values that are not provided by the other roles. For example, in case of an incomplete component specification, the resource demand of this component has to be estimated. Finally, they interpret the results of the QoS analyses and devise design alternatives, or they assist the software architects to do so. The roles of QoS analyst and software architect are also suitable to be incorporated by a single person.

The PCM supports QoS analysts and software archietcts to find better design alternatives by the automated architecture improvement approach PerOpteryx.

3.5.1 Automated Improvement of Architectures with PerOpteryx

Due to the large design space for non-trivial systems and many degrees of freedom, improving the architecture manually is an error-prone and tedious task. Isolated improvements of a single non-functional property, such as performance, can result in unexpected degradation for other quality properties, such as reliability, which are hard to determine and quantify by software architects manually. Also the improvement of quality properties might incur high cost. Thus, software architects and QoS analysts have to consider trade-offs between conflicting quality properties when designing a system.

Due to the trade-offs, it is difficult to specify non-functional requirements in advance. For example, even though it seems desirable at first to achieve a mean response time of three seconds for web-based system, this requirements may be sacrificed if the cost to achieve 3.3 seconds are considerably lower (for example because one server less is needed). Thus, the goal of an automated improvement approach cannot just be to find design alternatives that satisfy requirements, but it needs to improve several quality properties at once and provide the optimal trade-offs to the QoS analyst and software architect for decision making. Of course, if technical restrictions of the domain imply strict non-functional requirements, e.g. that airbags must open within fractions a of second, these are not subject to trade-offs.

While a completely synthesised design is certainly infeasible, many degrees of freedom remain in the software architecture after functional design and influence quality properties. For example, the component deployment, hardware sizing and possibly further configuration options can be adjusted. PerOpteryx manipulates instances of the PCM along these degrees of freedom and evaluates the resulting candidates for performance, reliability and cost. It uses the multi-objective evolutionary algorithm NSGA-II [39] internally.

With this approach, software architects do not have to search for alternative solutions manually. Instead, they can *focus on good solutions* automatically determined by our approach for trade-off decisions between multiple quality criteria. As the approach works on the architectural model level (as opposed to the performance model), architects can directly understand and use the automatically found solutions.

More details on PerOpteryx can be found in [40] and at [3].

3.5.1.1 Example Results

Example results of the automated improvements are shown in the following. The so-called business reporting system (BRS) has been optimised for performance, reliability and cost. The example system is described in detail in [40].

[3]https://sdqweb.ipd.kit.edu/wiki/PerOpteryx

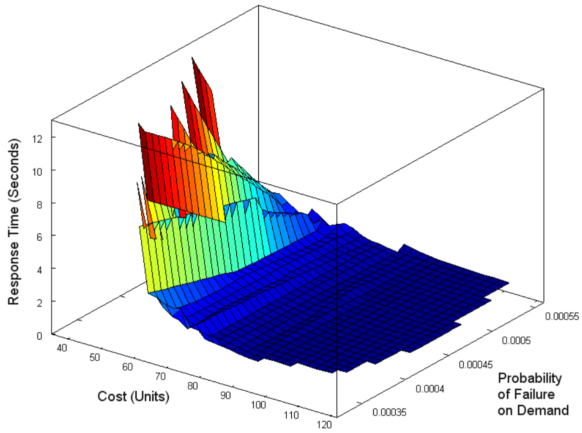


Figure 3.34: BRS system: 3D-Pareto front Performance vs. Reliability vs. Cost

The degrees of freedom that PerOpteryx explored were component selection, server processing rates, and component allocation:

Component selection is possible in this system as it contains several replaceable standard components. The `Web Server` as well as the `Database` can be realised using third party components. The software architect can choose among multiple functional equivalent components with different non-functional properties and cost. For the BRS, we have modelled two additional web servers and one additional database which have different performance and reliability properties, but also higher or lower cost than the components in the initial system.

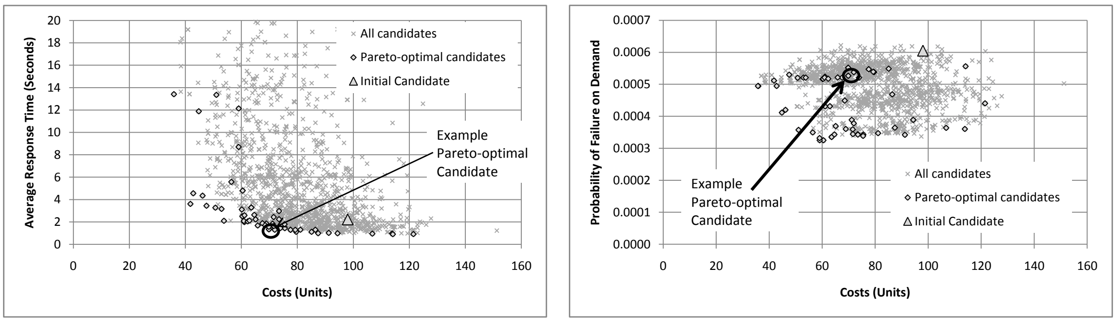
Server processing rates can be adjusted at multiple locations in the model as it contains 8 servers. It is expected that the overall performance of the system increases most significantly when using faster processing rates for highly utilised components. We assume here that the bounds for the processing rate are 1/2 of the initial rate (lower bound) and 2 times the initial rate (upper bound). Currently, the processing rate is modelled as a continuous variable.

Component allocation can be crucial for the non-functional properties and cost of the system. It could be possible to allocate multiple components on the same server without affecting the performance or reliability significantly. This could allow to remove some servers to save cost.

Figure 3.34 visualizes the three dimensional Pareto front for performance, reliability, and cost. Figure 3.35(a) shows the response time in seconds over the cost per candidate, while Figure 3.35(b) shows the probability of failure on demand over the cost per candidate. The software architect can use these results to make an informed trade-off decision among the different quality criteria.

In Figures 3.35(a) to 3.35(b), the 58 Pareto-optimal points are highlighted as thick squared marks. These points are not located at the borders of the candidate sets, as it would be the case for a two-dimensional set. Pareto-optimal points located within the set are superior to others in the third quality criterion not shown in the respective diagram (i.e., reliability in the first case and performance in the second case). For example, the highlighted candidate in Figure 3.35(a) has a higher response time and higher cost than some other candidates, but its reliability is superior to these other points.

We describe one of the found Pareto-optimal solutions in more detail. It is highlighted in Fig. 3.35(a) and Fig. 3.35(b) using circles. The response time of this solution is 1.34 seconds (initial candidate: 2.2 seconds), its POFOD is 0.0526 percent (initial candidate: 0.0605), and its cost are 69.83 units (initial



(a) Performance vs. Cost Tradeoff (b) Reliability vs. Cost Tradeoff

Figure 3.35: BRS system results

candidate: 98 units). Therefore it is superior to the initial candidate in all quality criteria.

3.5.1.2 Related Work

Rule-based approaches [41, 42, 43, 44, 45] translate known patterns, such as bottleneck removal or design diversity, into processable rules for manipulating architectural models. Applying these possibly contradicting rules can however easily lead to solutions locally but not globally optimal for a single quality property. Metaheuristic approaches [46, 47, 48, 49] encode the architecture model improvement as an optimisation problem and apply evolutionary algorithms, hill climbing, or simulated annealing [50]. Existing approaches in this direction either severely restrict expressiveness and degrees of freedom of the architectural model or are time-consuming, because of undirected search. See [40] for a more detailed discussion.

3.5.2 Future Work

We have planned several metamodel extensions to support QoS analysts. They should support adding required QoS values to model entities and also store the result of QoS predictions attached to corresponding model constructs. For example, the response time of an use case can be attached to a `UsageScenario` or the throughput of a resource to a `ProcessingResourceSpecification`. However, these modelling constructs have not been finalised and are subject to future work. So far, the QoS analyst is not explicitly supported by the PCM.

Chapter 4

Technical Reference

4.1 Package identifier

4.1.1 Package Overview

Provides a package for uniquely identifiable elements

4.1.2 Detailed Class Documentation

4.1.2.1 Class Identifier

Overview Inherit from this entity to make an element uniquely identifiable. Identifiers are not fixed to one realization. GUIDs are recommend. GUIDs are described in their own model. See GUIDModel (GUID.emx). Identifier implementations can be found in external projects only.

Class Properties Class `Identifier` has the following properties:

id : EString
Identifier attribute, in the default PCM implementation, this field is filled with a randomly generated UUID value

Constraints

idHasToBeUnique:

```
self.allInstances()->isUnique(p | p.id)
```

4.2 Package pcm

4.2.1 Package Overview

This package is the root package of all packages of the Palladio Component Model (PCM).
Note: This package does not contain any classes. Please see the contained sub-packages for classes.

4.3 Package pcm::allocation

4.3.1 Package Overview

All PCM entities related to model allocation

4.3.2 Package Diagrams

Figure Allocation Description Provides an overview on the relation between AllocationContext, resources and allocated entities (see Figure 4.1)

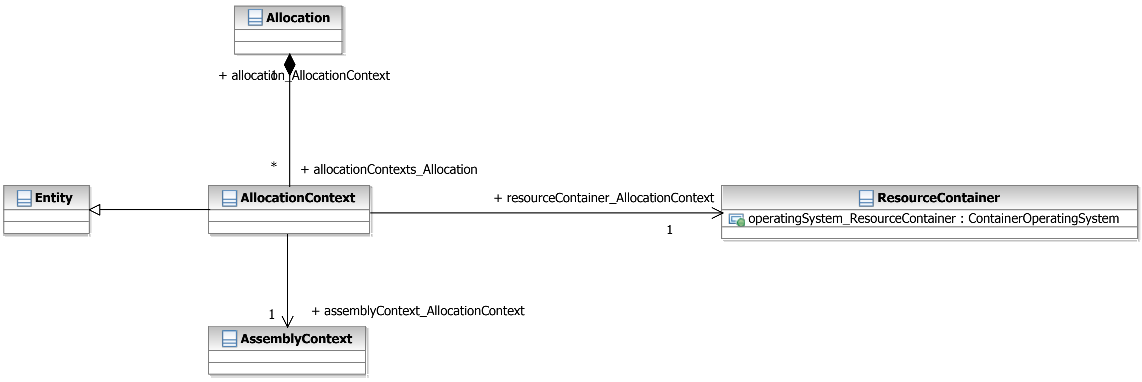


Figure 4.1: Allocation

4.3.3 Detailed Class Documentation

4.3.3.1 Class Allocation

Overview The allocation repository holding all available allocation contexts of a model.

Class Properties Class Allocation has the following properties:

allocationContexts_Allocation : AllocationContext [0..*]

system_Allocation : System

targetResourceEnvironment_Allocation : ResourceEnvironment [0..1]

Constraints

EachAssemblyContextWithinSystemHasToBeAllocatedExactlyOnce:

```
Each Assembly of BasicComponents and CompositeComponents used in the referenced System must be allocated.

Things are complicated by the introduction of SubSystems. Here, the Assembly of the SubSystem itself does not have to be allocated. If it is not allocated, all BasicComponents and CompositeComponents contained in this SubSystem (also transitively over several nested and not-allocated SubSystems) need to be allocated.

The constraint is realised wth a closure over the AssemblyContext contained in a ComposedStructure.
```

Parent Classes

- Entity (see section 4.7.3.2 on page 97)

4.3.3.2 Class `AllocationContext`

Overview Mapping between AssemblyContext and Resource. Sometimes referred to as "Deployment".

Class Properties Class `AllocationContext` has the following properties:

allocation_AllocationContext : Allocation

assemblyContext_AllocationContext : AssemblyContext

resourceContainer_AllocationContext : ResourceContainer

Parent Classes

- Entity (see section 4.7.3.2 on page 97)

4.4 Package `pcm::core`

4.4.1 Package Overview

This package contains the PCM Core meta-classes used throughout the PCM: entities carrying a globally unique ID (GUID), an abstract model for entities which provide and require interfaces, and an abstract model to describe entities composed from other entities.

4.4.2 Package Diagrams

Figure Entities Description Each entity is an Identifier and a NamedElement (see Figure 4.10)

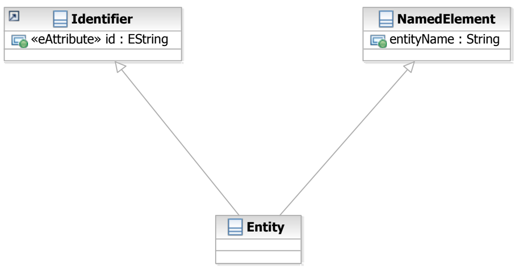


Figure 4.2: Entities

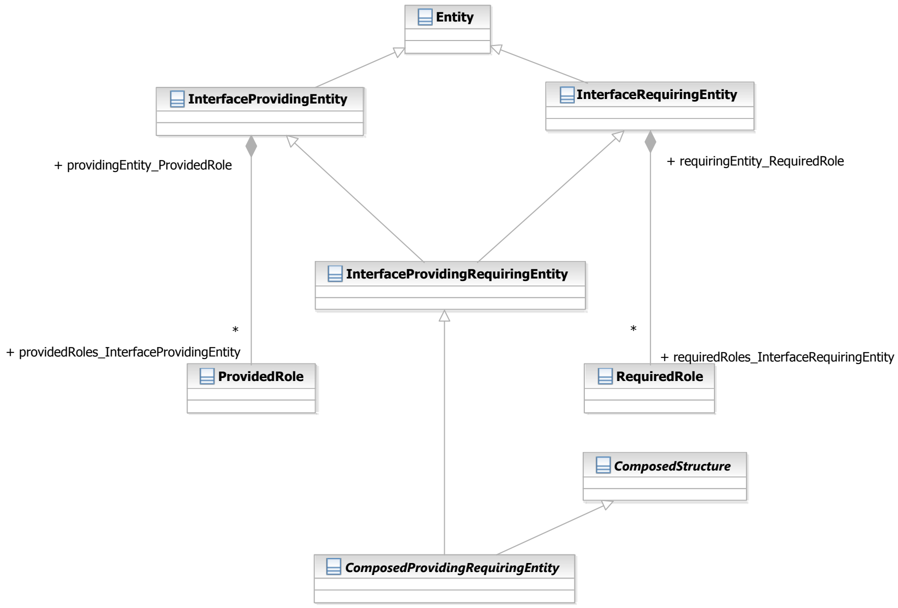


Figure 4.3: ProvidingRequiringEntities

Figure ProvidingRequiringEntities Description Overview on the core interface providing and requiring entities and their relation to ComposedStructure (see Figure 4.11)

Figure Composition Description The central element of this diagram is ComposedStructure. It holds n inner AssemblyContexts and n inner AssemblyConnectors. (see Figure 4.7)

Figure AssemblyConnector Description Overview on AssemblyConnector and AssemblyContext. An AssemblyConnector is a bidirectional link of two assembly contexts. (see Figure 4.8)

Figure Delegation Description <p> (see Figure 4.9)

4.4.3 Detailed Class Documentation

4.4.3.1 Class `PCMRandomVariable`

Overview Random variables are used to describe user and component behaviour. They allow not only constant values (e.g., 3 loop iterations), but also probabilistic values (e.g., 2 loop iterations with a probability of 0.4 and 3 loop iterations with a probability of 0.6). They are well-suited for capturing uncertainty when modelling systems during early development stages. Examples where developers may use random variables are: - Characterisations of Input Parameters: Describes the QoS relevant characteristics of parameters of component services. - Inter-Arrival Time: Describes how much time passes between the arrival of two subsequent users. - Think Time: Describes how much time passes between the execution of a user scenario and the start of the next execution of this scenario. - Loop Iteration Count: Describes the number of repetitions of a loop. - Guarded Branch Transitions: Used to determine whether to conditionally execute a certain behaviour.

Figure 4.4: Composition

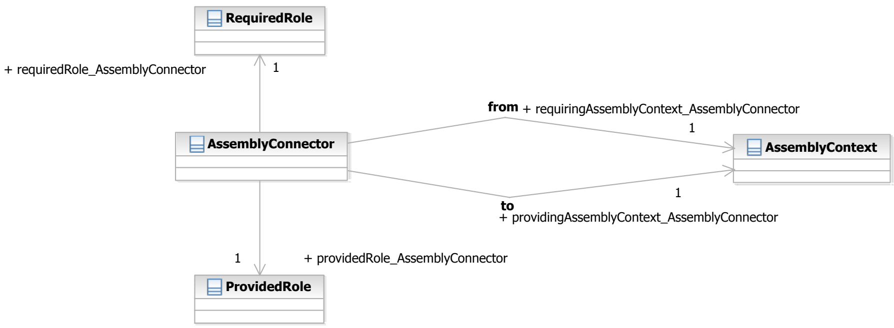


Figure 4.5: AssemblyConnector

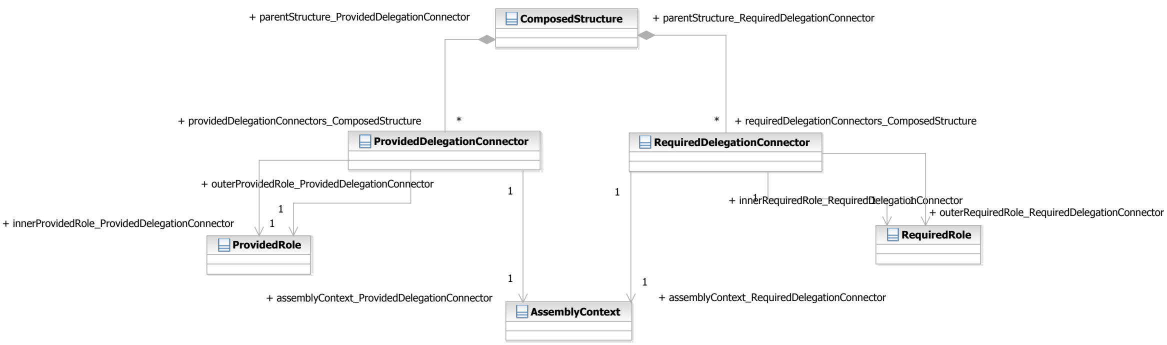


Figure 4.6: Delegation

PCMRandomVariable extends RandomVariable in a way, that the only type of variables available in the PCMRandomVariable are references to variable characterisations like a.NUMBER_OF_ELEMENTS. The corresponding editors ensure that the user can enter only valid expressions.

Class Properties Class `PCMRandomVariable` has the following properties:

closedWorkload_PCMRandomVariable : ClosedWorkload [0..1]

communicationLinkResourceSpecifcation_throughput_PCMRandomVariable : CommunicationLinkResource-Specification [0..1]

communicationLinkResourceSpecification_latency_PCMRandomVariable : CommunicationLinkResource-Specification [0..1]

delay_TimeSpecification : Delay [0..1]

guardedBranchTransition_PCMRandomVariable : GuardedBranchTransition [0..1]

loopAction_PCMRandomVariable : LoopAction [0..1]

loop_LoopIteration : Loop [0..1]

openWorkload_PCMRandomVariable : OpenWorkload [0..1]

parametricResourceDemand_PCMRandomVariable : ParametricResourceDemand [0..1]

passiveResource_capacity_PCMRandomVariable : PassiveResource [0..1]

processingResourceSpecification_processingRate_PCMRandomVariable : ProcessingResourceSpecification [0..1]

specifiedQoSAnnotation_SpecifiedExecutionTime : SpecifiedQoSAnnotation [0..1]

variableCharacterisation_Specification : VariableCharacterisation [0..1]

Constraints

SpecificationMustNotBeNULL:

```
not self.specification.oclIsUndefined() and self.specification <> ''
```

4.5 Package `pcm::core::composition`

4.5.1 Package Overview

A package holding all composable entities

4.5.2 Package Diagrams

Figure Composition Description The central element of this diagram is ComposedStructure. It holds n inner AssemblyContexts and n inner AssemblyConnectors. (see Figure 4.7)

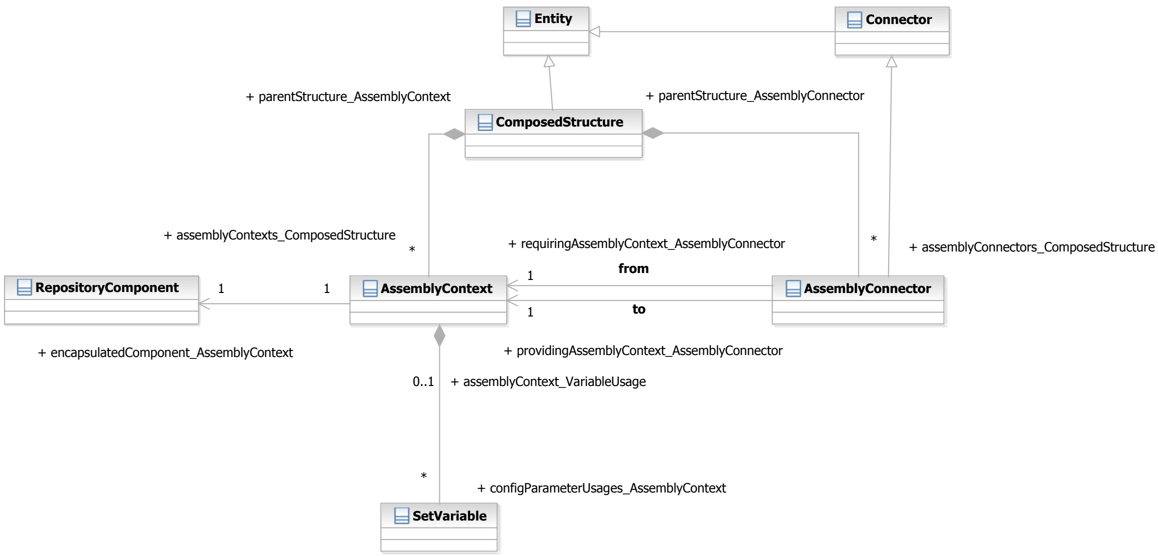


Figure 4.7: Composition

Figure AssemblyConnector Description Overview on AssemblyConnector and AssemblyContext. An AssemblyConnector is a bidirectional link of two assembly contexts. (see Figure 4.8)

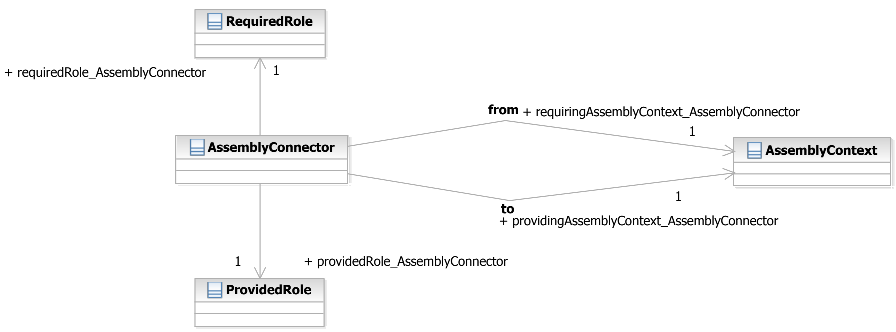


Figure 4.8: AssemblyConnector

Figure Delegation Description <p> (see Figure 4.9)

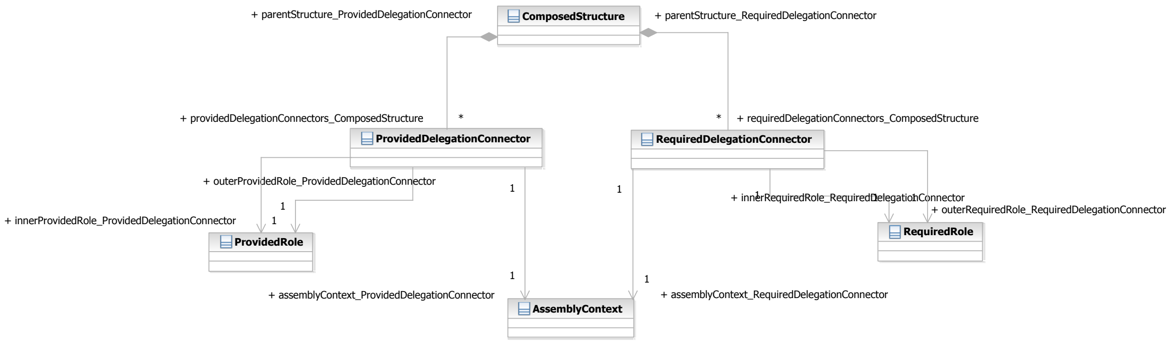


Figure 4.9: Delegation

4.5.3 Detailed Class Documentation

4.5.3.1 Class `AssemblyConnector`

Overview An AssemblyConnector is a bidirectional link of two assembly contexts. Intuitively, an AssemblyConnector connects a provided and a required interface of two different components. AssemblyContext must refer to the tuple (Role, AssemblyContext) in order to uniquely identify which component roles communicate with each other.

Class Properties Class `AssemblyConnector` has the following properties:

parentStructure_AssemblyConnector : ComposedStructure

providedRole_AssemblyConnector : ProvidedRole

providingAssemblyContext_AssemblyConnector : AssemblyContext

requiredRole_AssemblyConnector : RequiredRole

requiringAssemblyContext_AssemblyConnector : AssemblyContext

Constraints

AssemblyConnectorsReferencedProvidedRolesAndChildContextMustMatch:

```
An AssemblyConnector references an assembly context and a provided role on the ↘
  →provider side. This constraint ensures that the referenced provided role is really↘
  → available in the referenced assembly context.
```

AssemblyConnectorsReferencedRequiredRoleAndChildContextMustMatch:

```
An AssemblyConnector references an assembly context and a required role on the client ↘
  →side. This constraint ensures that the referenced required role is really ↘
  →available in the referenced assembly context.
```

AssemblyConnectorsReferencedInterfacesMustMatch:

```
The Interfaces references by this Connector must match. This means that either
1) the referenced providedRole and the referenced requiredRole refer to the same ↘
  →Interface (first part of the expression) or 2)  the Interface A referenced by the ↘
  →providedRole is a subtype of the Interface B referenced by the requiredRole as ↘
  →transitively defined by the parentInterface__Interface property. That means that ↘
  →either Interface A is the parentInterface__Interface of Interface B, or there is a↘
  → set of Interfaces
```

Parent Classes

- Connector (see section 4.6.2.1 on page 95) ,
- Entity (see section 4.7.3.2 on page 97)

4.5.3.2 Class AssemblyContext

Overview An AssemblyContext uniquely identifies an assembly instance of an AssemblyContext.

Class Properties Class `AssemblyContext` has the following properties:

configParameterUsages_AssemblyContext : SetVariable [0..∗]

encapsulatedComponent_AssemblyContext : RepositoryComponent

parentStructure_AssemblyContext : ComposedStructure

Parent Classes

- Entity (see section 4.7.3.2 on page 97)

4.5.3.3 Class ComposedStructure

Overview

Class Properties Class `ComposedStructure` has the following properties:

assemblyConnectors_ComposedStructure : AssemblyConnector [0..∗]

assemblyContexts_ComposedStructure : AssemblyContext [0..∗]

providedDelegationConnectors_ComposedStructure : ProvidedDelegationConnector [0..∗]

requiredDelegationConnectors_ComposedStructure : RequiredDelegationConnector [0..∗]

resourceRequiredDelegationConnectors_ComposedStructure : ResourceRequiredDelegationConnector [0..∗]

Constraints

MultipleConnectorsConstraint:

```
self.providedDelegationConnectors_ComposedStructure->forAll( c1, c2 | c1 <> c2 implies↘
→ c1.outerProvidedRole_ProvidedDelegationConnector <> c2.↘
→outerProvidedRole_ProvidedDelegationConnector)
```

MultipleConnectorConstraintForAssembyConnectors:

```
self.assemblyConnectors_ComposedStructure->forAll( c1, c2 | c1 <> c2 implies c1.↘
→requiredRole_AssemblyConnector <> c2.requiredRole_AssemblyConnector)
```

Parent Classes

- Entity (see section 4.7.3.2 on page 97)

4.5.3.4 Class `ProvidedDelegationConnector`

Overview A ProvidedDelegationConnector delegates incoming calls of provided roles to inner provided roles of encapsulated assembly contexts.

Class Properties Class `ProvidedDelegationConnector` has the following properties:

assemblyContext_ProvidedDelegationConnector : AssemblyContext

innerProvidedRole_ProvidedDelegationConnector : ProvidedRole

outerProvidedRole_ProvidedDelegationConnector : ProvidedRole

parentStructure_ProvidedDelegationConnector : ComposedStructure

Constraints

ProvidedDelegationConnector and the connected Component must be part of the same composite structure:

```
self.parentStructure_ProvidedDelegationConnector = self.↘
  →assemblyContext_ProvidedDelegationConnector.parentStructure_AssemblyContext
```

ComponentOfAssemblyContextAndInnerRoleProvidingComponentNeedToBeTheSame:

```
self.innerProvidedRole_ProvidedDelegationConnector.providingEntity_ProvidedRole = self↘
  →.assemblyContext_ProvidedDelegationConnector.encapsulatedComponent_AssemblyContext
```

Parent Classes

- DelegationConnector (see section 4.13.3.9 on page 119)

4.5.3.5 Class RequiredDelegationConnector

Overview A RequiredDelegationConnector delegates required roles of encapsulated assembly contexts to outer required roles .

Class Properties Class RequiredDelegationConnector has the following properties:

assemblyContext_RequiredDelegationConnector : AssemblyContext

innerRequiredRole_RequiredDelegationConnector : RequiredRole

outerRequiredRole_RequiredDelegationConnector : RequiredRole

parentStructure_RequiredDelegationConnector : ComposedStructure

Constraints

RequiredDelegationConnector and the connected Component must be part of the same composite structure:

```
self.parentStructure_RequiredDelegationConnector = self.↘
  →assemblyContext_RequiredDelegationConnector.parentStructure_AssemblyContext
```

ComponentOfAssemblyContextAndInnerRoleRequiringComponentNeedToBeTheSame:

```
self.innerRequiredRole_RequiredDelegationConnector.requiringEntity_RequiredRole = self↘
  →.assemblyContext_RequiredDelegationConnector.encapsulatedComponent_AssemblyContext
```

RequiringEntityOfOuterRequiredRoleMustBeTheSameAsTheParentOfTheRequiredDelegationConnector:

```
self.outerRequiredRole_RequiredDelegationConnector.requiringEntity_RequiredRole=self.↘
  →parentStructure_RequiredDelegationConnector
```

Parent Classes

- DelegationConnector (see section 4.13.3.9 on page 119)

4.5.3.6 Class `ResourceRequiredDelegationConnector`

Overview

Class Properties Class `ResourceRequiredDelegationConnector` has the following properties:

innerResourceRequiredRole_ResourceRequiredDelegationConnector : ResourceRequiredRole

outerResourceRequiredRole_ResourceRequiredDelegationConnector : ResourceRequiredRole

parentStructure_ResourceRequiredDelegationConnector : ComposedStructure

4.6 Package `pcm::core::connectors`

4.6.1 Package Overview

Package containing the abstract connector entity

4.6.2 Detailed Class Documentation

4.6.2.1 Class `Connector`

Overview Abstract superclass for all connectors.

Parent Classes

- Entity (see section 4.7.3.2 on page 97)

4.7 Package `pcm::core::entity`

4.7.1 Package Overview

This set of abstract meta-classes gives a conceptual view on interfaces, entities and their relationships: Two roles can be identified a software entity can take relative to an interface. Any entity can claim to implement the functionality specified in an interface as well as to request that functionality.

Base of the inheritance hierarchy are Identifier and NamedElement, both of which Entity and all inheriting classes are derived from.

The relationship of Entities and Interfaces is described with the three meta classes InterfaceProvidingEntity, InterfaceRequiringEntity, and InterfaceProvidingRequiringEntity. The abstract meta-class InterfaceProvidingEntity is inherited by all entities which can potentially offer interface implementations. Similarly, the meta-class InterfaceRequiringEntity is inherited by all entities which are allowed to request functionality offer by entities providing this functionality. InterfaceProvidingRequiringEntity inherits from both of them and thus combines their properties.

4.7.2 Package Diagrams

Figure Entities Description Each entity is an Identifier and a NamedElement (see Figure 4.10)

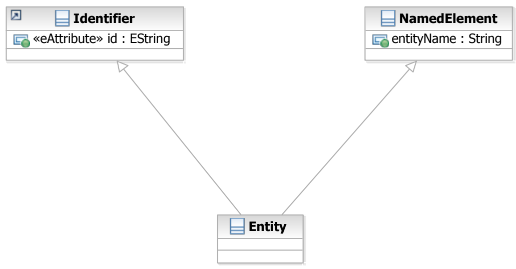


Figure 4.10: Entities

Figure ProvidingRequiringEntities Description Overview on the core interface providing and requiring entities and their relation to ComposedStructure (see Figure 4.11)

4.7.3 Detailed Class Documentation

4.7.3.1 Class `ComposedProvidingRequiringEntity`

Overview The ComposedProvidingRequiringEntity combines the properties of an InterfaceProvidingRequiringEntity and a ComposedStructure. It is inherited by all classes that, on the one hand, claim to implement the functionality specified in an interface as well as to request that functionality, and, on the other hand, are composed out of some inner entities.

Valid ComposedProvidingRequiringEntities need to have their ProvidedRoles bound to ProvidedRoles of the contained entities.

Prominent examples are System, SubSystem, and CompositeComponents

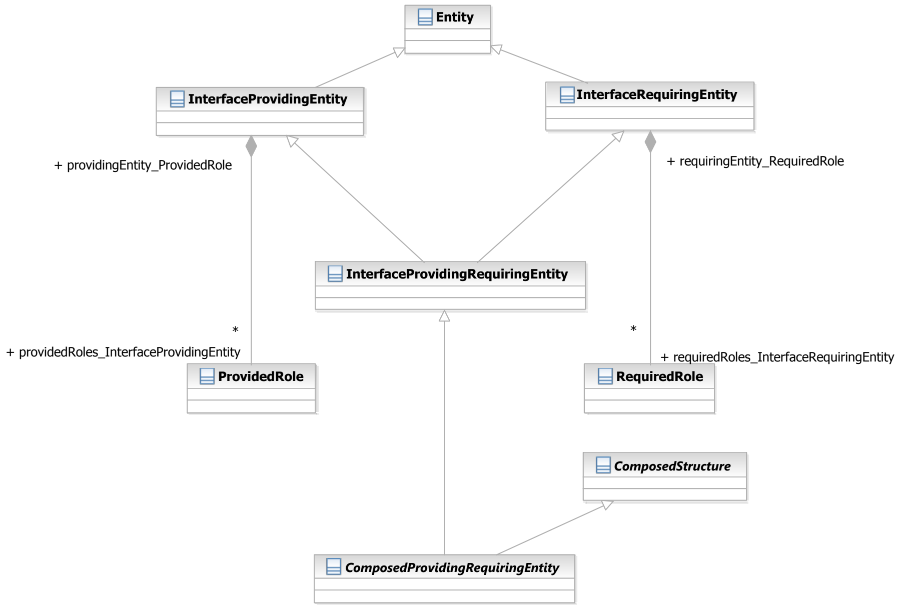


Figure 4.11: ProvidingRequiringEntities

Constraints

ProvidedRolesMustBeBound:

```
This constraint ensures that all outer provided roles of a system have a provided ↘
  →delegation conector that binds them to something. It does not check whether the ↘
  →binding is correct (inner role not null and matching interfaces).
```

Parent Classes

- ComposedStructure (see section 4.5.3.3 on page 92) ,
- InterfaceProvidingRequiringEntity (see section 4.7.3.4 on page 98)

4.7.3.2 Class Entity

Overview Entity is a meta class high up the PCM meta class hierarchy and represents all entities of the PCM that have both an id (see meta class Identifier) and a name (see meta class NamedEntity).

Parent Classes

- NamedElement (see section 4.7.3.6 on page 99)

4.7.3.3 Class `InterfaceProvidingEntity`

Overview All Entities that provide an Interface are represented by this class. Prominent inheriting classes are all component types, for example.

Two roles can be identified a software entity can take relative to an interface. Any entity can claim to implement the functionality specified in an interface as well as to request that functionality. This is reflected in the Palladio Component Model by a set of abstract meta-classes giving a conceptual view on interfaces, entities and their relationships. The abstract meta-class InterfaceProvidingEntity is inherited by all entities which can potentially offer interface implementations. Similarly, the meta-class InterfaceRequiringEntity is inherited by all entities which are allowed to request functionality offered by entities providing this functionality.

See also: Interface, ProvidedRole

Class Properties Class `InterfaceProvidingEntity` has the following properties:

providedRoles_InterfaceProvidingEntity : ProvidedRole [0..∗]

Parent Classes

- Entity (see section 4.7.3.2 on page 97)

4.7.3.4 Class `InterfaceProvidingRequiringEntity`

Overview This meta-class is inherited by classes that both require and provide an Interface. It thus combines the properties of InterfaceProvidingEntity and InterfaceRequiringEntity. Prominent inheriting classes are all component types, for example.

See also: Interface, ProvidedRole, RequiredRole, InterfaceProvidingEntity, InterfaceRequiringEntity

Parent Classes

- InterfaceProvidingEntity (see section 4.7.3.3 on page 98) ,
- InterfaceRequiringEntity (see section 4.7.3.5 on page 98) ,
- ResourceInterfaceRequiringEntity (see section 4.7.3.7 on page 99)

4.7.3.5 Class `InterfaceRequiringEntity`

Overview All Entities that require an Interface are represented by this class. Prominent inheriting classes are all component types, for example.

Two roles can be identified a software entity can take relative to an interface. Any entity can claim to implement the functionality specified in an interface as well as to request that functionality. This is reflected in the Palladio Component Model by a set of abstract meta-classes giving a conceptual view on interfaces, entities and their relationships. The abstract meta-class InterfaceRequiringEntity is inherited by all entities which are allowed to request functionality offered by entities providing this functionality. Similarly, the meta-classInterfaceProvidingEntity is inherited by all entities which can potentially offer interface implementations.

See also: Interface, RequiredRole

Class Properties Class `InterfaceRequiringEntity` has the following properties:

requiredRoles_InterfaceRequiringEntity : RequiredRole [0..$*$]

Parent Classes

- Entity (see section 4.7.3.2 on page 97)

4.7.3.6 Class `NamedElement`

Overview The NamedElement meta class is inherited by all PCM classes whose instances bear a name. Thus, the semantic of "bearing a name" is given to all inheriting classes, so that the name can be used in visualisations, for example.

Class Properties Class `NamedElement` has the following properties:

entityName : String

4.7.3.7 Class `ResourceInterfaceRequiringEntity`

Overview

Class Properties Class `ResourceInterfaceRequiringEntity` has the following properties:

resourceRequiredRoles_ResourceInterfaceRequiringEntity : ResourceRequiredRole [0..$*$]

Parent Classes

- Entity (see section 4.7.3.2 on page 97)

4.8 Package `pcm::parameter`

4.8.1 Package Overview

The parameter package allows to model data dependent performance characteristics of software systems. It is mainly used to specify performance dependencies on input and output parameters of single service calls. It can also be used to describe dependencies on the state of components by the use of component parameters. The latter describe stochastically a component state which does not change over time.

Parameters are described by the use of variable usages which on the one side contain a performance abstraction of the variable's value and on the other side the name of the variable for refering to the variable. Characterisations available include Structure (information on the data's internal structure like "sorted" or "unsorted" for an array), Number of Elements (size of a collection), Value (the actuall variable value), Bytesize (the variable's memory footprint), or type (the type of the variable in polymorphic cases).

Example for variable usages may be a.NUMBER_OF_ELEMENTS = 10 (array "a" contains 10 elements), tree.STRUCTURE = "balanced" (tree "tree" is a balanced tree), and so on.

4.8.2 Package Diagrams

Figure Parameter Package Overview Description The parameter package is used to model performance impacts of different types of data flow. It contains Named References to identify variables and characterisations to describe performance relevant meta-information on variables. (see Figure 4.12)

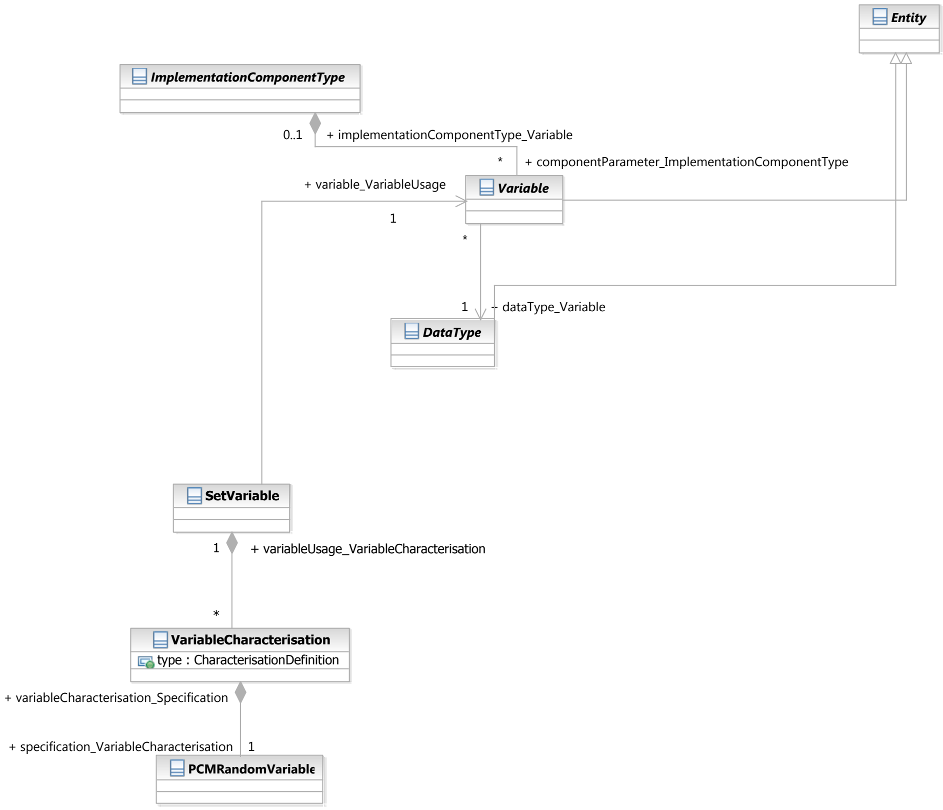


Figure 4.12: Parameter Package Overview

4.8.3 Detailed Class Documentation

4.8.3.1 Class `CharacterisedVariable`

Overview A characterised variable is a special variable which contains a performance abstraction of a data type. It can be transformed in a named reference and a variable characterisation. It has to end always with a variable characterisation type. Examples are "a.NUMBER_OF_ELEMENTS" or "array.STRUCTURE".

Class Properties Class `CharacterisedVariable` has the following properties:

characterisationDefinition : CharacterisationDefinition

variable : Variable

4.8.3.2 Class `SetVariable`

Overview SetVariable is used to set the characterisations of a Variable to certain values. Each of the set characterisations and the values are referenced via VariableCharacterisation. The deprecated name of this element was VariableUsage.

Class Properties Class `SetVariable` has the following properties:

assemblyContext_VariableUsage : AssemblyContext [0..1]

callAction_in_VariableUsage : CallAction [0..1]

callAction_out_VariableUsage : CallAction [0..1]

entryLevelSystemCall_InputParameterUsage : EntryLevelSystemCall [0..1]

entryLevelSystemCall_OutputParameterUsage : EntryLevelSystemCall [0..1]

setVariableAction_VariableUsage : SetVariableAction [0..1]

specifiedOutputParameterAbstraction_expectedExternalOutputs_VariableUsage : SpecifiedOutputParameterAbstraction [0..1]

synchronisationPoint_VariableUsage : SynchronisationPoint [0..1]

userData_VariableUsage : UserData [0..1]

variableCharacterisation_VariableUsage : VariableCharacterisation [0..*]
This association contains the information which abstract information on a specific variable is available. For example, whether we know something on the variable's value, its structure or memory footprint. There can be multiple characterisations of the same variable if more than one type of information is available.

variable_VariableUsage : Variable

4.8.3.3 Class `Variable`

Overview A named variable, e.g. a component, call, or return parameter.

Class Properties Class `Variable` has the following properties:

compositeDataType_Variable : CompositeDataType [0..1]

dataType_Variable : DataType

implementationComponentType_Variable : ImplementationComponentType [0..1]

signature_Parameter : Signature
This property navigates to the signature this parameter is a part of.

Parent Classes

- Entity (see section 4.7.3.2 on page 97)

4.8.3.4 Class `VariableCharacterisation`

Overview Variable characterisations store performance critical meta-information on a variable. The value of a characterisation is stored in the reference to the PCMRandomVariable.

For example, if a variable's value is used in a long running loop, the value of the variable is performance critical. For example, in "a.NUMBER_OF_ELEMENTS=10" the a is the name of the variable, NUMBER_OF_ELEMENTS is the name of the characterisation type and "10" would be the specification (as PCMRandomVariable) of the characterisation's value.

Class Properties Class `VariableCharacterisation` has the following properties:

specification_VariableCharacterisation : PCMRandomVariable
The specification contains the value of a variable characterisation. It is a stoachastic expression which may also contain references to other variable characterisations (that is the reason why it is a PCMRandomVariable).

type : CharacterisationDefinition
The type specifies the kind of the variable characterisation. There are 5 types available: STRUCTURE, NUMBER_OF_ELEMENTS, VALUE, BYTESIZE, and TYPE.

variableUsage_VariableCharacterisation : SetVariable

4.9 Package `pcm::protocol`

4.9.1 Package Overview

The PCM is prepared to support interface protocols. This package contains a protocol stub. Multiple protocols following different formalisms are supported by the PCM and distinguished by a protocol ID.

4.9.2 Detailed Class Documentation

4.9.2.1 Class `Protocol`

Overview A protocol is a set of calling sequences and can be optionally added to an interface. Protocols of provided interfaces specify the order in which services have to be called by clients. Protocols of required interfaces specify the actual order in which the component calls required services.

Besides finite state machines, different formalisms can be used to model protocols. The PCM does not restrict the protocol modelling formalisms. For example, Petri nets or regular expressions could model interface protocols. However, the choice of a formalism does influence possible analyses. For example, to check the interoperability of two components A and B, the language inclusion of the required protocol of A within the provided protocol of B has to be tested. The language inclusion is undecidable for Petri nets in the general case, so protocols modelled with Petri nets cannot be checked for interoperability. Notice, that although protocols are able to express the state of a component, interfaces themselves are stateless. The protocol state only depends on the component that implements the interface and is only present during component runtime. Components can provide/require multiple interfaces, but the PCM does not support protocols ranging over multiple interfaces (neither for provided nor required protocols). The complete state of a components consists of all its interface states. Restrictions on the complete state cannot be expressed in the PCM, as protocols can only be specified for single interfaces.

Class Properties Class `Protocol` has the following properties:

protocolTypeID : EString
Multiple protocols following different formalisms are supported by the PCM and distinguished by a protocol ID.

4.10 Package `pcm::qosannotations`

4.10.1 Package Overview

This package contains elements to specify fixed QoS attributes of system-external services.

4.10.2 Package Diagrams

Figure QosSpecification Description Overview on QoS specifications. (see Figure 4.13)

Figure Performance Description Performance QoS can either be specified execution times at the component level or system level. (see Figure 4.15)

4.10.3 Detailed Class Documentation

4.10.3.1 Class `QoSAnnotations`

Overview QoSAnnotations allow software architects and performance analysts to annotate Quality of Service (QoS) attributes to services (i.e., signatures of an interface). It is important to note that these annotations are specified and not derived. Usually the PCM uses the internal specification of a

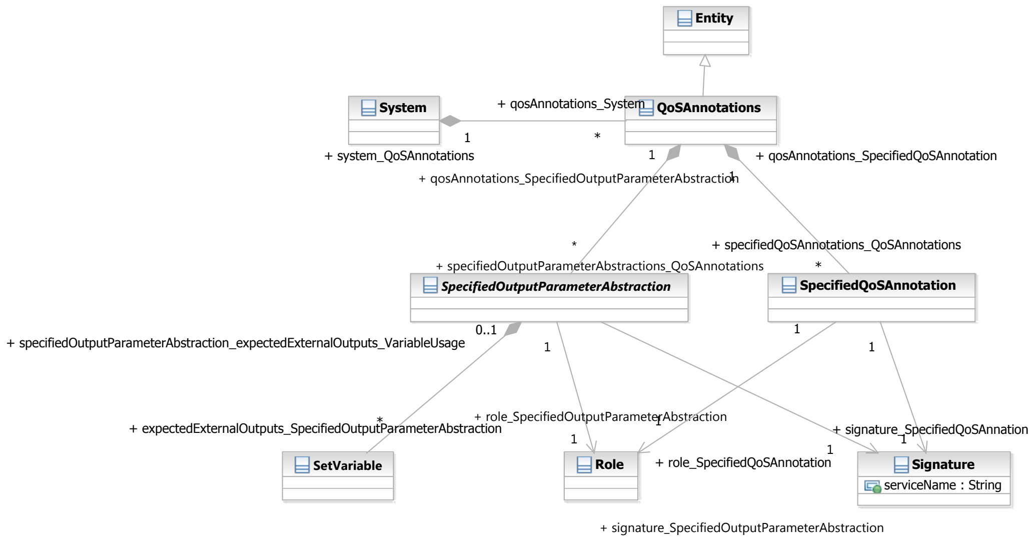


Figure 4.13: QosSpecification

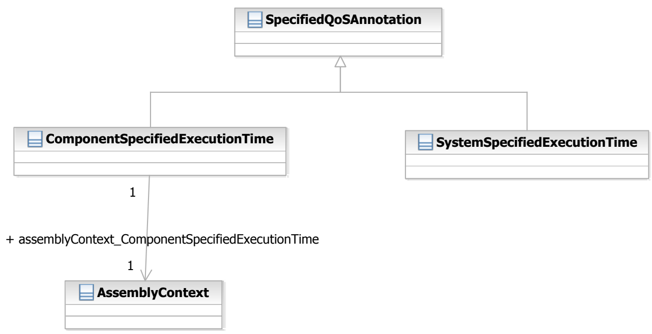


Figure 4.14: Performance

components behaviour (i.e., its RD-SEFFs) to determine its QoS. However, in a mixed top down and bottom up approach as favoured by the PCM, software architects have to combine components whose internals are not yet known with fully specified components. QoSAnnotations provide a first perforamnce (or reliability) abstraction of the services offered by a component using the SpecifiedQoSAnnotation. They furthermore define the output parameters of the services without describing its internal behviour.

Notes: - Should the association of QoSAnnotations to services not be in the class QoSAnnotation instead of SpecifiedQoSAnnotation and SpecifiedOutputParameterAbstraction separately?

Class Properties Class `QoSAnnotations` has the following properties:

specifiedOutputParameterAbstractions_QoSAnnotations : SpecifiedOutputParameterAbstraction [0..∗]

specifiedQoSAnnotations_QoSAnnotations : SpecifiedQoSAnnotation [0..∗]

system_QoSAnnotations : System

Parent Classes

- Entity (see section 4.7.3.2 on page 97)

4.10.3.2 Class `SpecifiedOutputParameterAbstraction`

Overview To specify the output parameters of a service (without associated RD-SEFF), software architects can associate a SpecifiedOutputParameterAbstraction to services (signature + role). Specified-OutputParameterAbstractions assign a single VariableUsage to that service that determines the output parameters in depency of its input parameters. Software architects can use stochastic expressions (package StoEx) to define the dependencies.

Note: - Is it actually possible to define the output in dependency on the input parameters?

Class Properties Class `SpecifiedOutputParameterAbstraction` has the following properties:

expectedExternalOutputs_SpecifiedOutputParameterAbstraction : SetVariable [0..∗]

qosAnnotations_SpecifiedOutputParameterAbstraction : QoSAnnotations

role_SpecifiedOutputParameterAbstraction : Role

signature_SpecifiedOutputParameterAbstraction : Signature

4.10.3.3 Class SpecifiedQoSAnnotation

Overview SpecifiedQoSAnnotations (as an abstract class) associate specified (see QoSAnnotation) QoS properties to services of components. A service is thereby determined by a Signature and a Role (i.e., an interface bound to a component). Whatever concrete QoS characteristic is specified, it has to be given in terms of a PCMRandomVariable which may depend on component or input parameters of the service.

Notes: - Is it correct that the PCMRandomvariable can depend on parameters? - How is the relation of the specified QoS to the input parameters established?

Class Properties Class `SpecifiedQoSAnnotation` has the following properties:

qosAnnotations_SpecifiedQoSAnnotation : QoSAnnotations

role_SpecifiedQoSAnnotation : Role

signature_SpecifiedQoSAnnation : Signature

specification_SpecifiedExecutionTime : PCMRandomVariable

4.11 Package pcm::qosannotations::performance

4.11.1 Package Overview

Performance aspects of QoS annotations.

4.11.2 Package Diagrams

Figure Performance Description Performance QoS can either be specified execution times at the component level or system level. (see Figure 4.15)

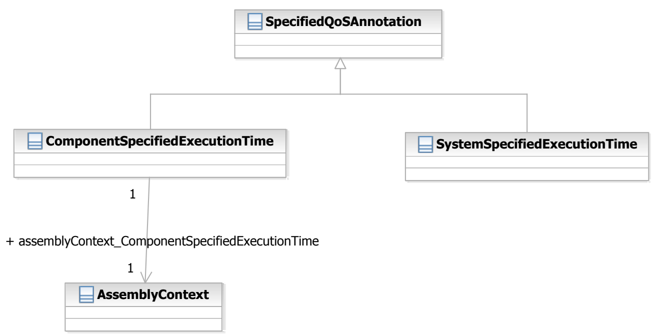


Figure 4.15: Performance

4.11.3 Detailed Class Documentation

4.11.3.1 Class ComponentSpecifiedExecutionTime

Overview The ComponentSpecifiedExecutionTime allows software architects (and performance analysts) to specify the response time of a service (signature + role) of a component. However, the response time is not given for the considered component in general, but the component in a specific context (i.e., in a specific hardware environment with specific external components connected) determined by the AssemblyContext. This allows software architects to include Provided- and CompleteComponentTypes into their software architecuture that still miss a description of their internals. Even though the internals are missing, performance predictions are still possible.

Note: - Is it actually the response time or total service demand specified here? -> I guess it should be response time. Otherwise, we would require also an assignment to resources. - I guess it's necessary to replace the association to the AssemblyContext by an association to an AllocationContext, since the Response time is heavily determined by the underlying hardware...

Class Properties Class `ComponentSpecifiedExecutionTime` has the following properties:

assemblyContext_ComponentSpecifiedExecutionTime : AssemblyContext

Parent Classes

- SpecifiedQoSAnnotation (see section 4.10.3.3 on page 106)

4.11.3.2 Class SystemSpecifiedExecutionTime

Overview The SystemSpecifiedExecutionTime allows software architect and performance analysts to specify the response time (distribution) of services called at the system boundaries. This allows abstracting from the system externals and restricts the focus to the software architecture under consideration.

Note: - That's the starting point for Performance-Kennlinien I guess...

Parent Classes

- SpecifiedQoSAnnotation (see section 4.10.3.3 on page 106)

4.12 Package pcm::qosannotations::reliability

4.12.1 Package Overview

Reliability aspects of QoS annotations.

4.12.2 Detailed Class Documentation

4.12.2.1 Class SpecifiedFailureProbability

Overview A SpecifiedFailureProbability of a service resembles its "Probability of Failure on Demand" (POFOD). Whenever the service is called, this values states the probability that it returns successfully.

For reliability prediction, the PCMRandomVariable specified in its superclass SpecifiedQoSAnnotation must evaluate to a real number between 0 and 1.

Class Properties Class `SpecifiedFailureProbability` has the following properties:

failureProbability : EDouble

Constraints

EnsureValidParameterRange:

```
self.failureProbability.oclAsType(Real) >= 0 and self.failureProbability.oclAsType(↘
  →Real) <= 1
```

Parent Classes

- SpecifiedQoSAnnotation (see section 4.10.3.3 on page 106)

4.13 Package `pcm::repository`

4.13.1 Package Overview

The main package contributing component types and interfaces.

4.13.2 Package Diagrams

Figure RepositoryTypeHierachy Description ComponentType hierarchy (see Figure 4.16)

Figure CompositeComponent Description Inheritance relation of CompositeComponent (see Figure 4.17)

Figure Interface Description Overview on the Interface structure (see Figure 4.18)

Figure Datatype Description Overview on the structure of DataTypes (see Figure 4.19)

Figure RepositoryContainments Description Overview on repository containments (see Figure 4.20)

Figure PassiveResources Description BasicComponents can carry PassiveResources (see Figure 4.21)

Figure RepositoryComponent Description <p> (see Figure 4.22)

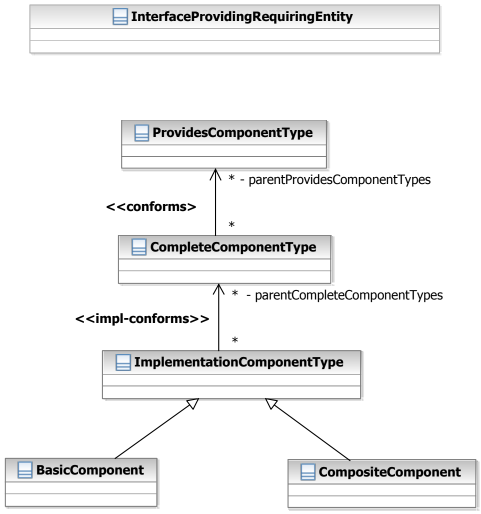


Figure 4.16: RepositoryTypeHierachy

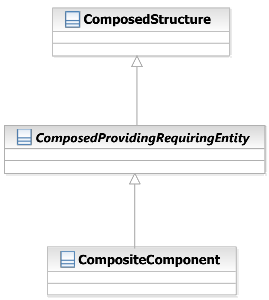


Figure 4.17: CompositeComponent

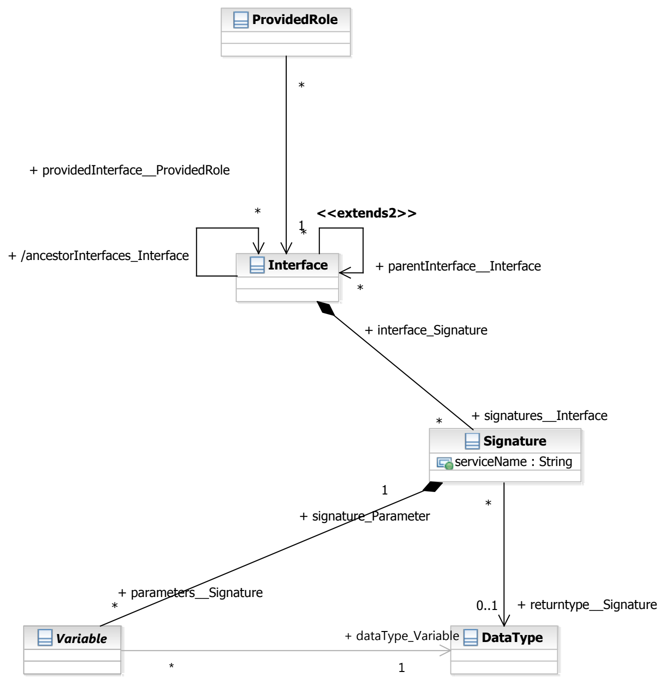


Figure 4.18: Interface

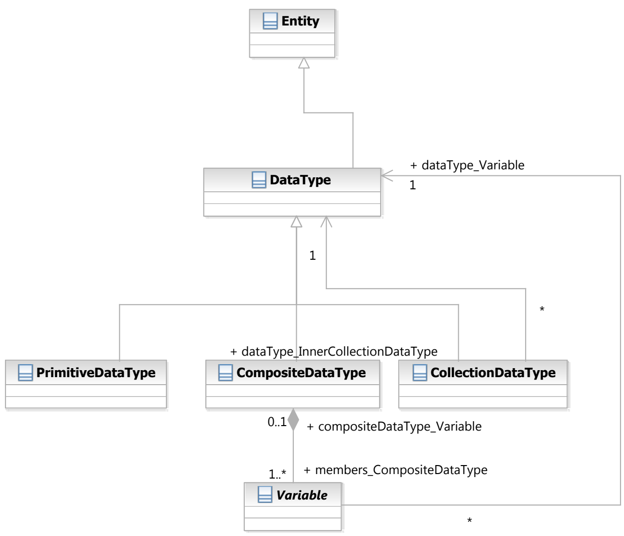


Figure 4.19: Datatype

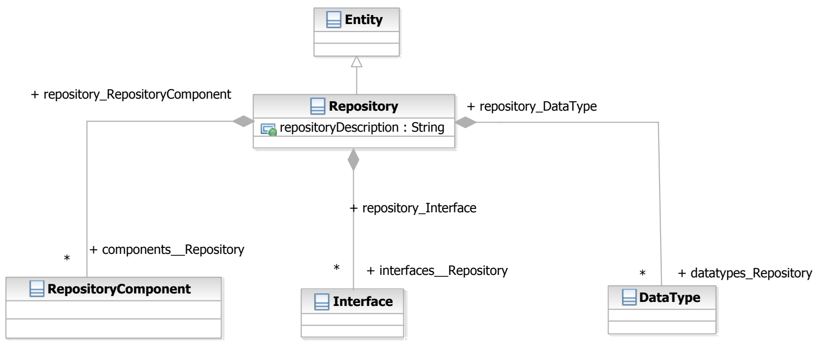


Figure 4.20: RepositoryContainments

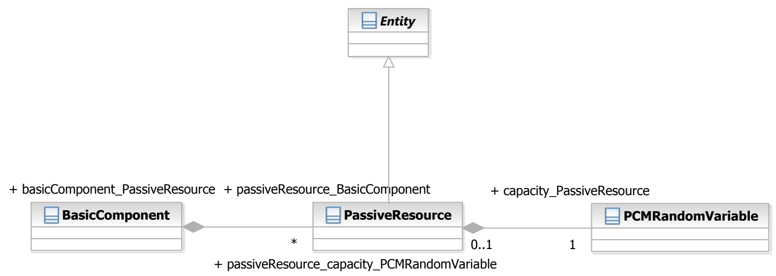


Figure 4.21: PassiveResources

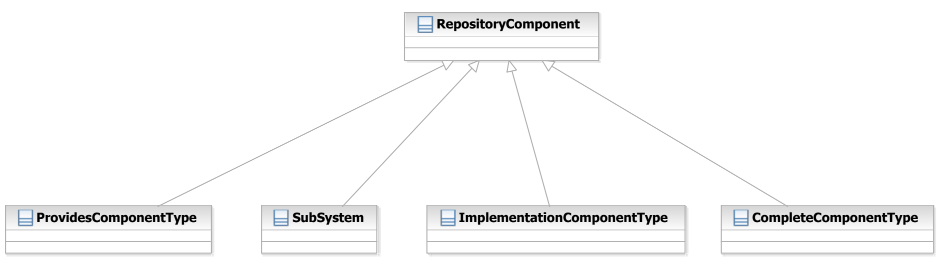


Figure 4.22: RepositoryComponent

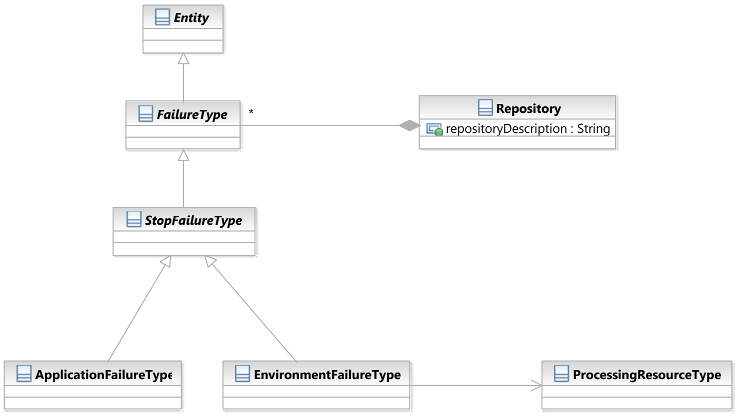


Figure 4.23: FailureTypes

Figure FailureTypes Description <p> (see Figure 4.23)

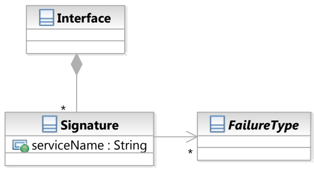


Figure 4.24: InterfaceFailureSpecifications

Figure InterfaceFailureSpecifications Description

Figure DatatypeCharacterisation Description

4.13.3 Detailed Class Documentation

4.13.3.1 Class ApplicationFailureType

Overview Application failures originate in the application itself. InteralActions may specify application failures.

Parent Classes

- StopFailureType (see section 4.13.3.25 on page 129)

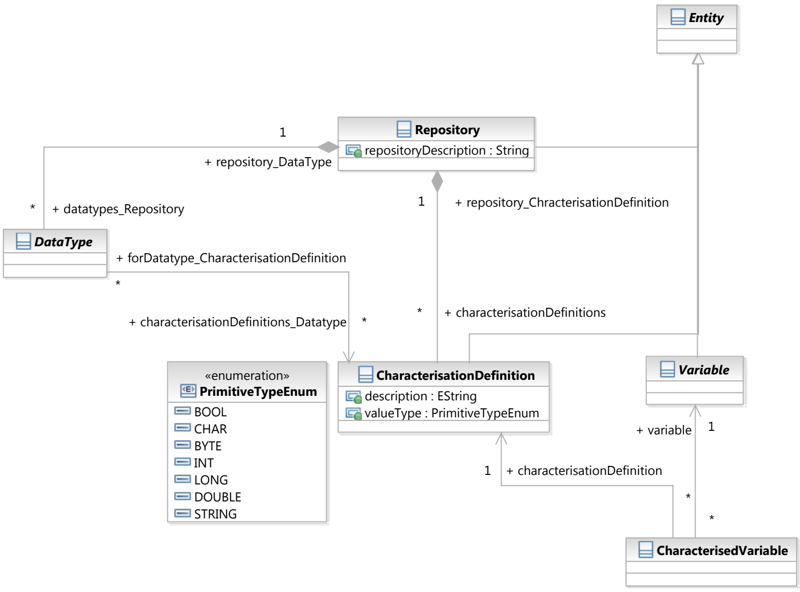


Figure 4.25: DatatypeCharacterisation

4.13.3.2 Class `BasicComponent`

Overview This entity represents a black-box component implementation. Basic components are atomic building blocks of a software architecture. They cannot be further subdivided into smaller components and are built from scratch, i.e, not by assembling other components. Component developers specify basic components by associating interfaces to them in a providing or requiring role.

Class Properties Class `BasicComponent` has the following properties:

passiveResource_BasicComponent : PassiveResource [0..∗]
This property represents the passive resources, e.g., semaphores, that are owned by this basic component.

serviceEffectSpecifications__BasicComponent : ServiceEffectSpecification [0..∗]
This property contains the service effect specification for services provided by this basic component.

Constraints

NoSeffTypeUsedTwice:

```
self.serviceEffectSpecifications__BasicComponent->forAll(p1, p2 |
  p1 <> p2 implies (p1.describedService__SEFF = p2.describedService__SEFF implies p1.↘
   →seffTypeID <> p2.seffTypeID))
```

ProvideSameInterfaces As Implementation Type:

```
-- BC has to provide the same interfaces like the implementationComponentType (if set)↘
  → #
if
         -- apply constraint only for non-empty ImplementationComponentTypes of a BC #
        self.parentCompleteComponentTypes->notEmpty()
then
        --own interface IDs:
    self.providedRoles_InterfaceProvidingEntity->collect(pr : ProvidedRole | pr.↘
      →providedInterface__ProvidedRole.id)->asSet()
    =
    --complete type interface IDs:
    self.parentCompleteComponentTypes->collect(pr | pr.↘
      →providedRoles_InterfaceProvidingEntity.providedInterface__ProvidedRole.id)->↘
      →asSet()
else
        true
endif
```

Require Same Interfaces As Implementation Type:

```
-- BC has to require the same interfaces like the implementationComponentType (if set)↘
  → #
if
         -- apply constraint only for non-empty ImplementationComponentTypes of a BC #
        self.parentCompleteComponentTypes->notEmpty()
then
        --own interface IDs:
    self.requiredRoles_InterfaceRequiringEntity->collect(rr : RequiredRole | rr.↘
      →requiredInterface__RequiredRole.id)->asSet()
    =
    --complete type interface IDs:
    self.parentCompleteComponentTypes->collect(rr | rr.↘
      →requiredRoles_InterfaceRequiringEntity.requiredInterface__RequiredRole.id)->↘
      →asSet()
else
        true
endif
```

Parent Classes

- ImplementationComponentType (see section 4.13.3.13 on page 120)

4.13.3.3 Class CharacterisationDefinition

Overview Defines a characterisation. An exemplary characterisation is: name BYTESIZE, description 'Size of the data type in memory in bytes', and value type LONG.

Class Properties Class `CharacterisationDefinition` has the following properties:

description : EString

repository_ChracterisationDefinition : Repository

valueType : PrimitiveTypeEnum

Parent Classes

- Entity (see section 4.7.3.2 on page 97)

4.13.3.4 Class CollectionDataType

Overview This entity represents a collection data type, e.g.,. a list, array, set, of items of the referenced type.

Class Properties Class `CollectionDataType` has the following properties:

dataType_InnerCollectionDataType : DataType

Parent Classes

- DataType (see section 4.13.3.8 on page 118)

4.13.3.5 Class CompleteComponentType

Overview Complete (Component) types abstract from the realisation of components. They only contain provided and required roles omitting the components? internal structure, i.e., the service effect specifications or assemblies. Thus, complete types represent a black box view on components. Leaving the implementation open allows for a higher flexibility of software architects and defines substitutability in the PCM.

Class Properties Class `CompleteComponentType` has the following properties:

parentProvidesComponentTypes : ProvidesComponentType [0..∗]

Constraints

AtLeastOneInterfaceHasToBeProvidedOrRequiredByAUsefullCompleteComponentType:

```
(
        self.oclIsTypeOf(CompleteComponentType)
        or
        self.oclIsTypeOf(ImplementationComponentType)
        or
        self.oclIsTypeOf(CompositeComponent)
        or
        self.oclIsTypeOf(BasicComponent)
)
implies
(
        self.providedRoles_InterfaceProvidingEntity->size() >= 1
        or
        self.requiredRoles_InterfaceRequiringEntity->size() >= 1
)
```

providedInterfacesHaveToConformToProvidedType2:

```
-- CompleteTypes provided Interfaces have to be a superset
-- of ProvidesComponentType provided Interfaces #
--
-- ACCx are used to accumulate Sets/Bags; usually only the very inner ACCx is used at ↘
  →all.
--
-- Recursive Query for parent Interface IDs
-- see "lpar2005.pdf" (Second-order principles in specification languages for Object-↘
  →Oriented Programs; Beckert, Tretelman) pp. 11 #
--let parentInterfaces : Bag(Interface) =
--        self.providedRoles->iterate(r : ProvidedRole; acc2 : Bag(Interface) = Bag{} |
--                acc2->union(r.providedInterface.parentInterface->asBag()) -- asBag ↘
  →required to allow Set operations #
--        ) in
--let anchestorInterfaces : Bag(Interface) =
--        self.providedRoles->iterate(r : ProvidedRole; acc4 : Bag(Interface) = Bag{} |
--                acc4->union(r.providedInterface.parentInterface->asBag()) -- asBag ↘
  →required to allow Set operations #
--        )->union( -- union with anchestors found in former recursion #
--                self.providedRoles->iterate(r : ProvidedRole; acc6 : Bag(Interface) = ↘
  →Bag{} |
--                        acc6->union(r.providedInterface.parentInterface.↘
  →anchestorInterfaces) --already Set/Bag
--                )
--        ) in
--        -- Directly provided anchestorInterfaces need to be a superset of provided ↘
  →interfaces of Supertype #
```

```
--      anchestorInterfaces.identifier.id->includesAll(
--              self.parentProvidesComponentTypes->iterate(pt : ProvidesComponentType;↘
  → acc1 : Bag(String) = Bag{} |
--                      pt.providedRoles->iterate(r : ProvidedRole; acc2 : Bag(String)↘
  → = Bag{} |
--                              acc2->union(r.providedInterface.identifier.id->asBag()↘
  →) -- asBag required to allow Set operations #
--                      )
--              )
--      )
true
```

Parent Classes

- RepositoryComponent (see section 4.13.3.20 on page 126)

4.13.3.6 Class CompositeComponent

Overview Composite components are special implementation component types, which are composed from inner components. Component developers compose inner components within composite components with assembly connectors. An assembly connector binds a provided role with a required role. To access the inner components, composite components themselves provide or require interfaces. A delegation connector binds a provided (required) role of the composite component with an inner component provided (required) role. A composite component may contain other composite components, which are also themselves composed out of inner components. This enables building arbitrary hierarchies of nested components. Like a basic component, a composite component may contain a SEFF. However, this SEFF is not specified manually by the composite component developer, but can be computed by combining the SEFFs of the inner components.

Constraints

ProvideSameInterfaces:

```
-- CC has to provide the same interfaces like the implementationComponentType (if set)↘
  → (same OCL code like BC)#
if
         -- apply constraint only for non-empty ImplementationComponentTypes of a BC #
        self.parentCompleteComponentTypes->notEmpty()
then
        --own interface IDs:
    self.providedRoles_InterfaceProvidingEntity->collect(pr : ProvidedRole | pr.↘
      →providedInterface__ProvidedRole.id)->asSet()
    =
    --complete type interface IDs:
    self.parentCompleteComponentTypes->collect(pr | pr.↘
      →providedRoles_InterfaceProvidingEntity.providedInterface__ProvidedRole.id)->↘
      →asSet()
```

```
else
        true
endif
```

RequireSameInterfaces:

```
-- CC has to require the same interfaces like the implementationComponentType (if set)↘
  → (same OCL code like BC) #
if
         -- apply constraint only for non-empty ImplementationComponentTypes of a BC #
        self.parentCompleteComponentTypes->notEmpty()
then
        --own interface IDs:
    self.requiredRoles_InterfaceRequiringEntity->collect(rr : RequiredRole | rr.↘
      →requiredInterface__RequiredRole.id)->asSet()
    =
    --complete type interface IDs:
    self.parentCompleteComponentTypes->collect(rr | rr.↘
      →requiredRoles_InterfaceRequiringEntity.requiredInterface__RequiredRole.id)->↘
      →asSet()
else
        true
endif
```

Parent Classes

- ComposedProvidingRequiringEntity (see section 4.7.3.1 on page 96) ,
- ImplementationComponentType (see section 4.13.3.13 on page 120)

4.13.3.7 Class CompositeDataType

Overview This entity represents a complex data type containing which is composed by Variables. This construct is common in higher programming languages as record, struct, or class.

Class Properties Class CompositeDataType has the following properties:

members_CompositeDataType : Variable [1..∗]

Parent Classes

- DataType (see section 4.13.3.8 on page 118)

4.13.3.8 Class DataType

Overview This entity represents a data type that can be stored in a repository and used for specification and modeling of interface signatures or component parameters. All valid characterisations of a data type a referenced.

Class Properties Class `DataType` has the following properties:

characterisationDefinitions_Datatype : CharacterisationDefinition [0..∗]

repository_DataType : Repository
This property specifies the repository to which this data type belongs.

Constraints

DataTypeMustNotHaveAvailableCharacterisationsWithIdenticalNameToBeParsableAsCode:

```
self.availableCharacterisationDefinitions_Datatype.name->asSet()->size() = self.↘
→availableCharacterisationDefinitions_Datatype->size()
```

Parent Classes

- Entity (see section 4.7.3.2 on page 97)

4.13.3.9 Class `DelegationConnector`

Overview This entity represents a delegation connector, i.e., connector used for connecting a provided/required role of a component woth provided/reqiured port of its subcomponent.

Parent Classes

- Connector (see section 4.6.2.1 on page 95)

4.13.3.10 Class `EnvironmentFailureType`

Overview Environment failure are caused by unavailable or malfunctioning resources. The EnvironmentFailureType represents a failure of a specific resource type.

Class Properties Class `EnvironmentFailureType` has the following properties:

processingresourcetype : ProcessingResourceType

Constraints

Exactly One Resource:

```
self.processingresourcetype <> null
```

Parent Classes

- StopFailureType (see section 4.13.3.25 on page 129)

4.13.3.11 Class ExceptionType

Overview This entity represents a type of an exception.

Class Properties Class `ExceptionType` has the following properties:

exceptionMessage : EString
This property holds the text message of the exception.

exceptionName : EString
This property denotes the name of the exception. In addition to the exception message, this is another piece of information that can be used for identification of the exception that has appeared.

4.13.3.12 Class FailureType

Overview Represents failures that can occur in a software system.

Class Properties Class `FailureType` has the following properties:

repository_FailureType : Repository

Parent Classes

- Entity (see section 4.7.3.2 on page 97)

4.13.3.13 Class ImplementationComponentType

Overview This entity represents an abstraction of a component, where both sets of provided and required interfaces as well as the implementation is visible. It fully specifies the component type. The specification of the internal structure depends on the way the component is realised. In general, components can either be implemented from the scratch or composed out of other components. In the first case, the implemented behaviour of each provided service needs to be specified with a service effect specification (SEFF) to describe the component?s abstract internal structure. We refer to such components as basic components, since they form the basic building blocks of a software architecture. On the other hand, developers can use existing components to assemble new, composite components.

Class Properties Class `ImplementationComponentType` has the following properties:

componentParameter_ImplementationComponentType : Variable [0..∗]

parentCompleteComponentTypes : CompleteComponentType [0..∗]

Constraints

RequiredInterfacesHaveToConformToCompleteType:

```
-- ImplementationTypes required Interfaces have to be a subset
-- of CompleteComponentType required Interfaces #
--
-- ACCx are used to accumulate Sets/Bags; usually only the very inner ACCx is used at ↘
  →all.
--
-- Recursive Query for parent Interface IDs
-- see "lpar2005.pdf" (Second-order principles in specification languages for Object-↘
  →Oriented Programs; Beckert, Tretelman) pp. 11 #
--let parentInterfaces : Bag(Interface) =
--      self.parentCompleteComponentTypes->iterate(pt : CompleteComponentType; acc1 : ↘
  →Bag(Interface) = Bag{} |
--              acc1->union(pt.requiredRoles->iterate(r : RequiredRole; acc2 : Bag(↘
  →Interface) = Bag{} |
--                      acc2->union(r.requiredInterface.parentInterface->asBag()) -- ↘
  →asBag required to allow Set operations #
--              ))
--      ) in
--let anchestorInterfaces : Bag(Interface) =
--      self.parentCompleteComponentTypes->iterate(pt : CompleteComponentType; acc3 : ↘
  →Bag(Interface) = Bag{} |
--              acc3->union(pt.requiredRoles->iterate(r : RequiredRole; acc4 : Bag(↘
  →Interface) = Bag{} |
--                      acc4->union(r.requiredInterface.parentInterface->asBag()) -- ↘
  →asBag required to allow Set operations #
--              ))
--      )->union( -- union with anchestors found in former recursion #
--              self.parentCompleteComponentTypes->iterate(pt : CompleteComponentType;↘
  → acc5 : Bag(Interface) = Bag{} |
--                      acc5->union(pt.requiredRoles->iterate(r : RequiredRole; acc6 :↘
  → Bag(Interface) = Bag{} |
--                              acc6->union(r.requiredInterface.parentInterface.↘
  →anchestorInterfaces) --already Set/Bag
--                      ))
--              )
--      ) in
-- Directly required interfaces need to be a subset of required anchestorInterfaces of↘
  → Supertype #
--anchestorInterfaces.identifier.id->includesAll(
--      self.requiredRoles->iterate(p : RequiredRole; acc7 : Bag(String) = Bag{} |
--              acc7->union(p.requiredInterface.identifier.id->asBag())
--      )
--)
true
```

providedInterfacesHaveToConformToCompleteType:

```
-- ### EXACT COPY FROM ABOVE ###
-- ImplementationComponentTypes provided Interfaces have to be a superset
-- of CompleteComponentType provided Interfaces #
--
-- ACCx are used to accumulate Sets/Bags; usually only the very inner ACCx is used at ↘
  →all.
--
-- Recursive Query for parent Interface IDs
-- see "lpar2005.pdf" (Second-order principles in specification languages for Object-↘
  →Oriented Programs; Beckert, Tretelman) pp. 11 #
--let parentInterfaces : Bag(Interface) =
--      self.providedRoles->iterate(r : ProvidedRole; acc2 : Bag(Interface) = Bag{} |
--              acc2->union(r.providedInterface.parentInterface->asBag()) -- asBag ↘
  →required to allow Set operations #
--      ) in
--let anchestorInterfaces : Bag(Interface) =
--      self.providedRoles->iterate(r : ProvidedRole; acc4 : Bag(Interface) = Bag{} |
--              acc4->union(r.providedInterface.parentInterface->asBag()) -- asBag ↘
  →required to allow Set operations #
--      )->union( -- union with anchestors found in former recursion #
--              self.providedRoles->iterate(r : ProvidedRole; acc6 : Bag(Interface) = ↘
  →Bag{} |
--                      acc6->union(r.providedInterface.parentInterface.↘
  →anchestorInterfaces) --already Set/Bag
--              )
--      ) in
        -- Directly provided anchestorInterfaces need to be a superset of provided ↘
          →interfaces of Supertype #
--      anchestorInterfaces.identifier.id->includesAll(
--              self.parentProvidesComponentTypes->iterate(pt : ProvidesComponentType;↘
  → acc1 : Bag(String) = Bag{} |
--                      pt.providedRoles->iterate(r : ProvidedRole; acc2 : Bag(String)↘
  → = Bag{} |
--                              acc2->union(r.providedInterface.identifier.id->asBag()↘
  →) -- asBag required to allow Set operations #
--                      )
--              )
--      )
true
```

Parent Classes

- RepositoryComponent (see section 4.13.3.20 on page 126)

4.13.3.14 Class `Interface`

Overview This entity models the interface as a set of signatures representing services provided or required by a component. An interface is an abstraction of piece of software (a software entity) which should contain a sufficient amount of information for a caller to understand and finally request the realised functionality from any entity claiming to offer the specified functionality. Note that this implies, that the specification of the interface also has to contain a sufficient amount of information for the implementer to actually implement the interface. Due to the inherent need of an interface to abstract the behaviour of the software entity not in all cases there is sufficient information provided to use or implement an interface in an unambiquious way.

Interfaces can exist on their own, i.e., without any entity requesting or implementing the specified functionality. Two roles can be identified a software entity can take relative to an interface. Any entity can either claim to implement the functionality specified in an interface or to request that functionality. This is reflected in the Palladio Component Model by a set of abstract meta-classes giving a conceptual view on interfaces, entities, and their relationships.

Class Properties Class `Interface` has the following properties:

ancestorInterfaces_Interface : Interface [0..∗]
This property represents the set of all parent interfaces, from which this interface inherits. All means not just the direct one.

parentInterface__Interface : Interface [0..∗]
This property represents the interfaces from which this interface directly inherits.

protocols__Interface : Protocol [0..∗]
This property represents the protocol bound to this interfaces, i.e., the way, in the sense of the order, the services of this interfaces are allowed to be called.

repository_Interface : Repository
This property represents the repository where this interface is stored.

signatures__Interface : Signature [0..∗]
This property represents the set of signatures of which the interface consists.

Constraints

NoProtocolTypeIDUsedTwice:

```
self.protocols__Interface->forAll(p1, p2 |
p1.protocolTypeID <> p2.protocolTypeID)
```

SignaturesHaveToBeUniqueForAnInterface:

```
-- full signature has to be unique
-- (use of ocl-tupels) #
let sigs : Bag(
        -- parameters: Sequence of DataType, NOT name #
        -- exceptions have not to be considered #
        Tuple(returnType : DataType, serviceName : String, parameters : Sequence(↘
          →DataType) )
) =
self.signatures__Interface->collect(sig : Signature |
        Tuple{
                returnType : DataType = sig.returntype__Signature,
                serviceName : String = sig.serviceName,
                parameters : Sequence(DataType) = sig.parameters__Signature.↘
                  →datatype__Parameter
        }
)
in
sigs->isUnique(s|s)
```

Parent Classes

- Entity (see section 4.7.3.2 on page 97)

4.13.3.15 Class PassiveResource

Overview This entity represents a passive resource, e.g., a semaphore.

Class Properties Class PassiveResource has the following properties:

basicComponent_PassiveResource : BasicComponent

capacity_PassiveResource : PCMRandomVariable
This property holds the capacity of this passive resource.

Parent Classes

- Entity (see section 4.7.3.2 on page 97)

4.13.3.16 Class PrimitiveDataType

Overview This entity represents a primitive data type. Examples are integer, string, double, or any java class.

Parent Classes

- DataType (see section 4.13.3.8 on page 118)

4.13.3.17 Class ProvidedRole

Overview This entity represents the provided interfaces. The PCM uses the association of an interface to a component to determine its role. Provided roles list the interfaces offered by a component.

Class Properties Class `ProvidedRole` has the following properties:

providedInterface__ProvidedRole : Interface
This property represents the corresponding interface that is provided by this role.

providingEntity_ProvidedRole : InterfaceProvidingEntity
This property represents the providing entity that is providing the interface associated with this role.

Parent Classes

- Role (see section 4.13.3.23 on page 127)

4.13.3.18 Class ProvidesComponentType

Overview Provided (Component) Types abstract a component to its provided interfaces, leaving its requirements and implementation details open. So, provided types subsume components which offer the same functionality, but with different implementations. As different implementations might require different services from the environment, provided types omit required interfaces. Provided types allow software architects to focus on a component?s functionality and introduce weak substitutability to the PCM. Using provided types, software architects can draft ideas on how functionality can be partitioned among different components without worrying about their implementation. In the initial phases of architectural design, it often does not make sense to arrange all details of a component, since most of them depend on the actual implementation and thus need to be specified by component developers. As during this phase the actual implementation is unknown, also the required interfaces of a component cannot be stated. However, software architects can still pre-evaluate a software architecture containing provided-types. This gives rough estimates about the quality of the build software system and defines QoS requirements for the component implementation.

Constraints

AtLeastOneInterfaceHasToBeProvidedByAUsefullProvidesComponentType:

```
self.oclIsTypeOf(ProvidesComponentType)
implies
self.providedRoles_InterfaceProvidingEntity->size() >= 1
```

Parent Classes

- RepositoryComponent (see section 4.13.3.20 on page 126)

4.13.3.19 Class Repository

Overview The repository entity allows storing components, data types, and interfaces to be fetched and reused for construction of component instances as well as new component types.

Class Properties Class `Repository` has the following properties:

characterisationDefinitions : CharacterisationDefinition [0..∗]

components__Repository : RepositoryComponent [0..∗]
This property represents the provides component types stored in the repository.

datatypes_Repository : DataType [0..∗]
This property represents the data types stored in the repository.

failureTypes : FailureType [0..∗]

interfaces__Repository : Interface [0..∗]
This property represents the interfaces stored in the repository.

repositoryDescription : String [0..1]
This property represents a description of the repository.

Parent Classes

- Entity (see section 4.7.3.2 on page 97)

4.13.3.20 Class RepositoryComponent

Overview Abstract superclass of all component types which can be part of a component repository

Class Properties Class `RepositoryComponent` has the following properties:

repository_RepositoryComponent : Repository
This property represents the repository where this entity is stored.

Parent Classes

- InterfaceProvidingRequiringEntity (see section 4.7.3.4 on page 98)

4.13.3.21 Class `RequiredRole`

Overview This entity represents a required interface.

Class Properties Class `RequiredRole` has the following properties:

requiredInterface__RequiredRole : Interface
This property represents the interfaces that is required by this role.

requiringEntity_RequiredRole : InterfaceRequiringEntity
This property represents the interface requiring entity that requires this interface.

Parent Classes

- Role (see section 4.13.3.23 on page 127)

4.13.3.22 Class `ResourceRequiredRole`

Overview Required role for resource interface access of a component

Class Properties Class `ResourceRequiredRole` has the following properties:

requiredInterface_ResourceRequiredRole : Interface

resourceRequiringEntity_ResourceRequiredRole : ResourceInterfaceRequiringEntity

Parent Classes

- Role (see section 4.13.3.23 on page 127)

4.13.3.23 Class `Role`

Overview This entity represents an abstraction of an interface role.

Parent Classes

- Entity (see section 4.7.3.2 on page 97)

4.13.3.24 Class `Signature`

Overview This entity represents the signature of a method, i.e., its parameters, exception declarations, return type, etc.

Every service of an interface has a unique signature, like void doSomething(int a). A PCM signature is comparable to a method signature in programming languages like C#, Java or the OMG IDL.

It contains:

* A type of the return value or void (no return value),
* an identifier naming the service,
* an ordered set of parameters (0..*).Each parameter is a tuple of a datatype and an identifier (which is unique across the parameters). Additionally, the modifiers in, out, and inout (with its OMG IDL semantics) can be used for parameters, and
* an unordered set of exceptions.

A signature has to be unique for an interface through the tupel (identifier, order of parameters). An interface has a list of 1..* signatures (interfaces associate 1..* signatures, not the other way around). A signature is assigned to exactly one interface. However, different interfaces can define equally named signatures. If, for example, void doIt() is defined for interface A and B, void doIt() is not identical in both interfaces.

Failure that may occur inside external services must be specified at the service signatures. This way components that use this services may implement failure handling without knowing the internal behaviour of the connected component.

Class Properties Class `Signature` has the following properties:

exceptions__Signature : ExceptionType [0..∗]
This property represents the list of exceptions declared by this signature.

failureType : FailureType [0..∗]

interface_Signature : Interface
This property represents the interface that contains the method with this signature.

parameters__Signature : Variable [0..∗]
This property represents the list of parameters of the corresponding method.

returntype__Signature : DataType [0..1]
This property represents the return type of the corresponding method.

serviceName : String
This property represents the service name realized by this method.

Constraints

Parameter Names Have To Be Unique For A Signature:

```
self.parameters__Signature->isUnique(p : Parameter |
        p.parameterName
)
```

4.13.3.25 Class StopFailureType

Overview Represents failures with stop semantics. Such failures lead to direct interuption of the current control flow and passing the failure information up the calling hierarchy.

Parent Classes

- FailureType (see section 4.13.3.12 on page 120)

4.14 Package pcm::resourceenvironment

4.14.1 Package Overview

Package of entities representing the execution environment of a component based software system

4.14.2 Package Diagrams

Figure ResourceEnvironment Description Overview on all elements of the resource environment (see Figure 4.26)

Figure RandomVariableSpecifications Description Overview on usages of the PCM random variable (see Figure 4.27)

4.14.3 Detailed Class Documentation

4.14.3.1 Class CommunicationLinkResourceSpecification

Overview Throughput and performance specification of linking resources

Class Properties Class `CommunicationLinkResourceSpecification` has the following properties:

communicationLinkResourceType_CommunicationLinkResourceSpecification : CommunicationLinkResourceType

failureProbability : EDouble
Specifies the probability that a service call over this communication link fails. The failure could be due to message loss or overload, for example.

latency_CommunicationLinkResourceSpecification : PCMRandomVariable

linkingResource_CommunicationLinkResourceSpecification : LinkingResource

throughput_CommunicationLinkResourceSpecification : PCMRandomVariable

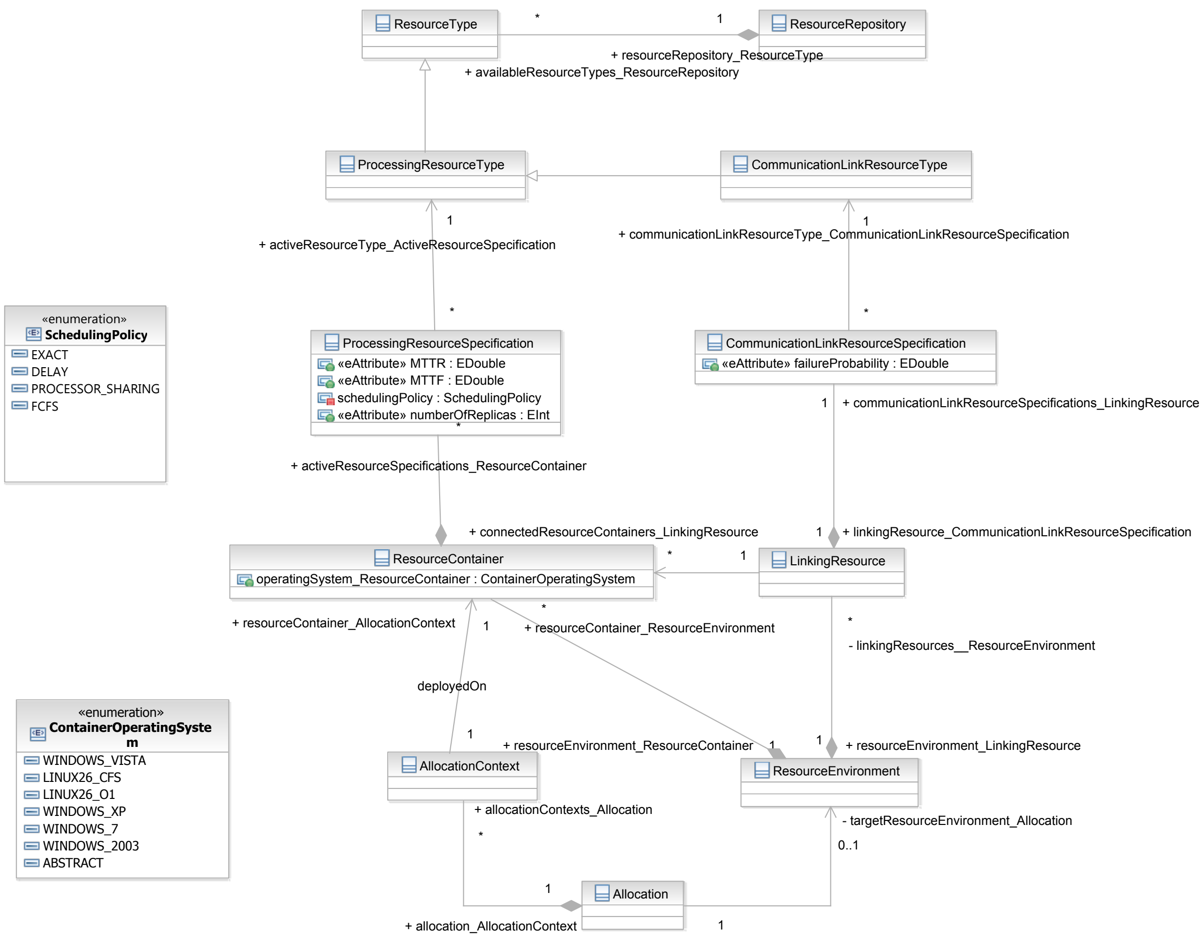


Figure 4.26: ResourceEnvironment

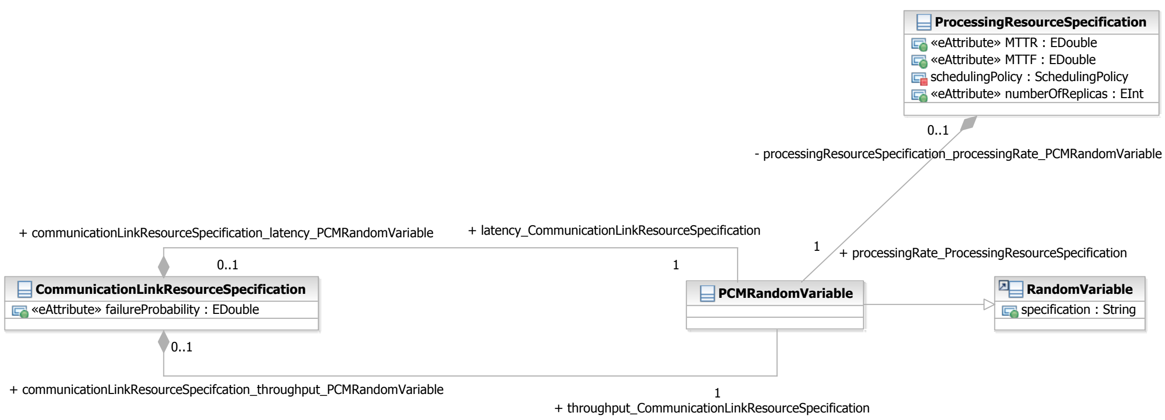


Figure 4.27: RandomVariableSpecifications

4.14.3.2 Class LinkingResource

Overview Model element representing communication links like LAN, WAN, WiFi etc.

Class Properties Class `LinkingResource` has the following properties:

communicationLinkResourceSpecifications_LinkingResource : CommunicationLinkResourceSpecification

connectedResourceContainers_LinkingResource : ResourceContainer [0..∗]

resourceEnvironment_LinkingResource : ResourceEnvironment

Parent Classes

- Entity (see section 4.7.3.2 on page 97)

4.14.3.3 Class ProcessingResourceSpecification

Overview Performance specification of processing resources (e.g. processing rate, scheduling policy)

Class Properties Class `ProcessingResourceSpecification` has the following properties:

MTTF : EDouble
The Mean Time To Failure (MTTF) of a physical resource is the expected timespan from the start of its usage until breakdown.

MTTR : EDouble
The Mean Time To Repair (MTTR) of a physical resource is the expected timespan from breakdown of this physical resource to its repair or replacement.

activeResourceType_ActiveResourceSpecification : ProcessingResourceType

numberOfReplicas : EInt
Specifies the actual number of processors of the processing resource.
In terms of the queueing theory, the number of processors corresponds to the number of servers of a service center. Thus, the attribute allows to specify a multi-server queue, i.e., one queue with multiple servers.

processingRate_ProcessingResourceSpecification : PCMRandomVariable

resourceContainer_ProcessingResourceSpecification : ResourceContainer

schedulingPolicy : SchedulingPolicy

4.14.3.4 Class `ResourceContainer`

Overview UML-like container of a number of processing resources (e.g. hardware server)

Class Properties Class `ResourceContainer` has the following properties:

activeResourceSpecifications_ResourceContainer : ProcessingResourceSpecification [0..∗]

operatingSystem_ResourceContainer : ContainerOperatingSystem

resourceEnvironment_ResourceContainer : ResourceEnvironment

Parent Classes

- Entity (see section 4.7.3.2 on page 97)

4.14.3.5 Class `ResourceEnvironment`

Overview Repository element of the resource environment

Class Properties Class `ResourceEnvironment` has the following properties:

linkingResources__ResourceEnvironment : LinkingResource [0..∗]

resourceContainer_ResourceEnvironment : ResourceContainer [0..∗]

Parent Classes

- NamedElement (see section 4.7.3.6 on page 99)

4.15 Package `pcm::resourcetype`

4.15.1 Package Overview

Package containing all resource types supported by the PCM

4.15.2 Package Diagrams

Figure Resources Description Overview on resource demands and processing resource type relations (see Figure 4.28)

Figure Units Description ResourceTypes are UnitCarryingElements (see Figure 4.29)

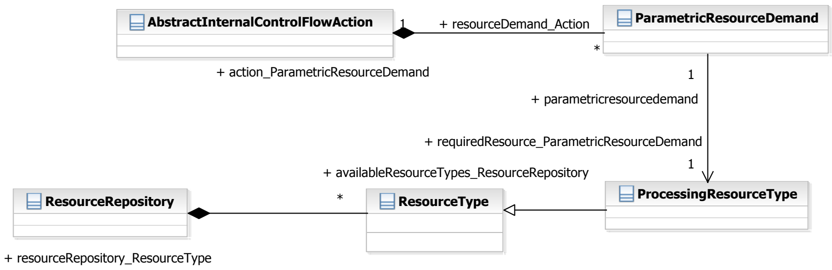


Figure 4.28: Resources

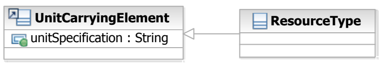


Figure 4.29: Units

4.15.3 Detailed Class Documentation

4.15.3.1 Class `CommunicationLinkResourceType`

Overview ResourceType representing communication links like, LAN, WAN, WiFi etc.

Parent Classes

- ProcessingResourceType (see section 4.15.3.2 on page 133)

4.15.3.2 Class `ProcessingResourceType`

Overview ResourceType representation of CPU.

Parent Classes

- ResourceType (see section 4.15.3.4 on page 133)

4.15.3.3 Class `ResourceRepository`

Overview Extendable repository of resource types of the PCM. The resource type repository is intentionally left open to support arbitrary resources in the future

Class Properties Class `ResourceRepository` has the following properties:

availableResourceTypes_ResourceRepository : ResourceType [0..$*$]

4.15.3.4 Class `ResourceType`

Overview Abstract superclass of any resource

Class Properties Class `ResourceType` has the following properties:

resourceRepository_ResourceType : ResourceRepository

Parent Classes

- Entity (see section 4.7.3.2 on page 97)

4.16 Package `pcm::seff`

4.16.1 Package Overview

Package containing the abstract behaviour model of components

4.16.2 Package Diagrams

Figure Resource Demand Description <p> (see Figure 4.30)

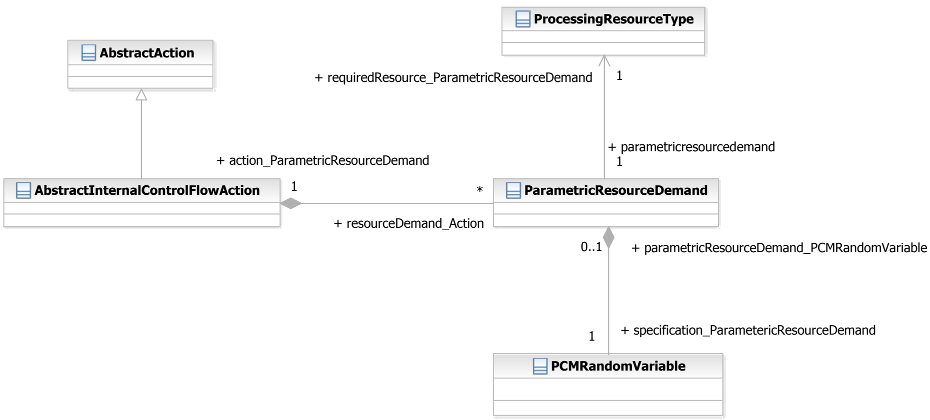


Figure 4.30: Resource Demand

Figure Actions Description ExternalCallAction and InternalCallAction in the PCM (see Figure 4.31)

Figure SEFF Description Relations between ServiceEffectSpecification, BasicComponent and ResourceDemandingBehaviour (see Figure 4.32)

Figure Loop Behaviour Description Overview on loop behaviour (see Figure 4.33)

Figure Parameter Usage Description <p> (see Figure 4.34)

Figure Branch Description Overview on the abstract branch transition (see Figure 4.35)

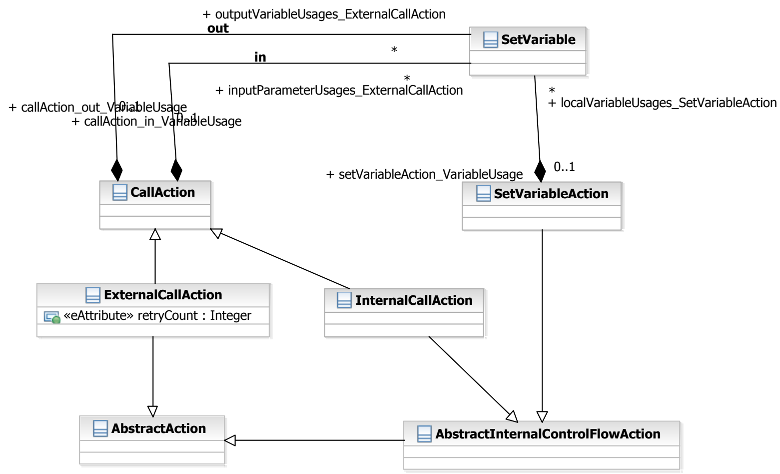


Figure 4.31: Actions

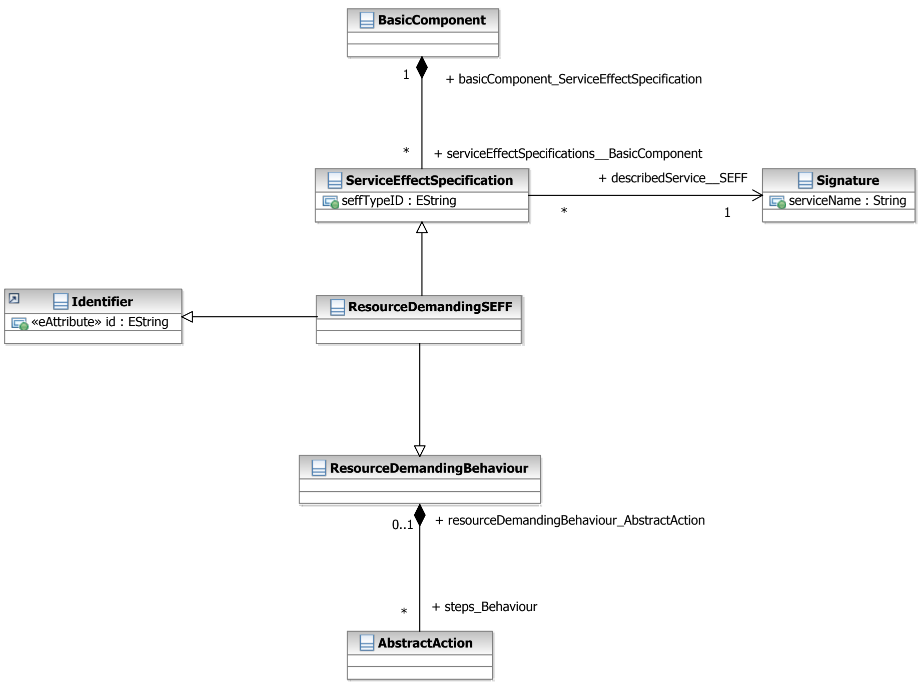


Figure 4.32: SEFF

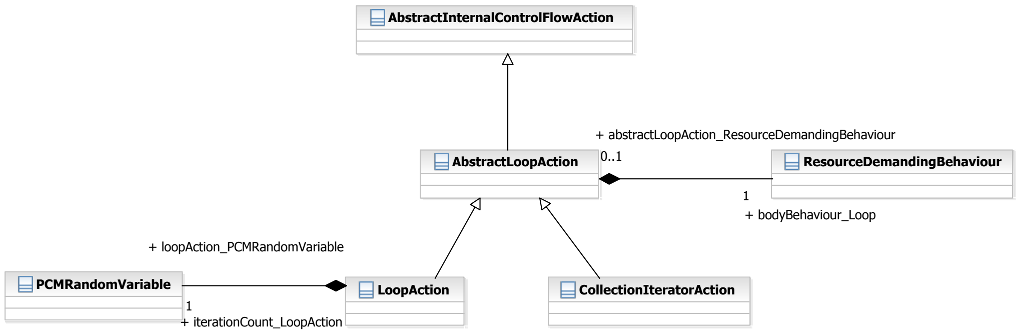


Figure 4.33: Loop Behaviour

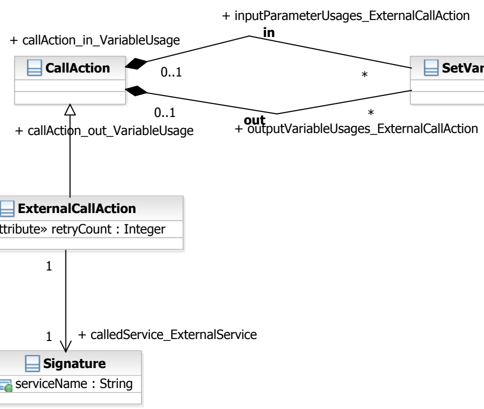


Figure 4.34: Parameter Usage

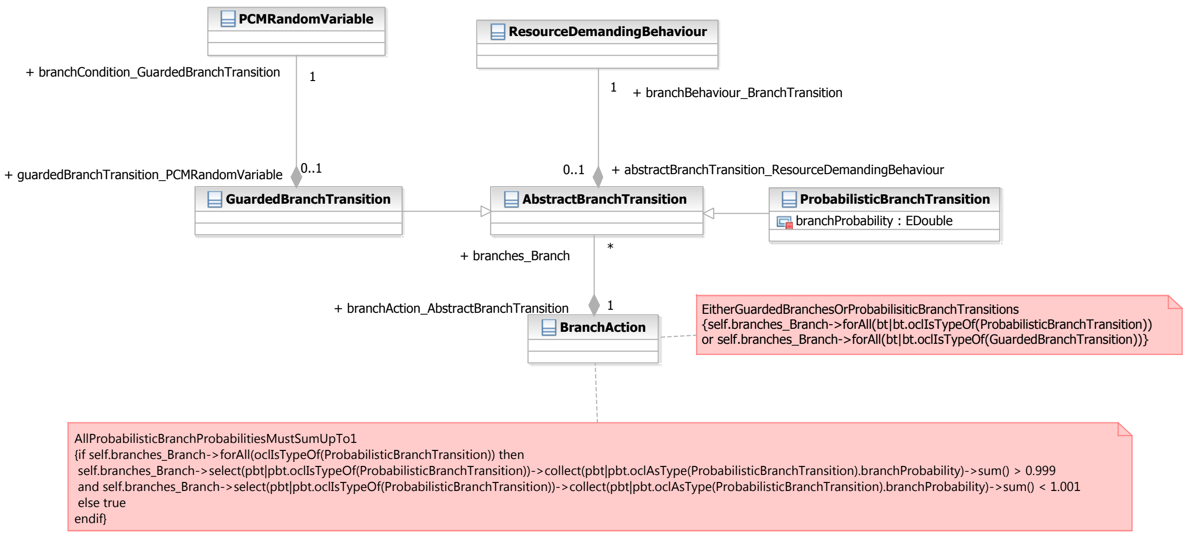


Figure 4.35: Branch

Figure BehaviourOverview Description Overview on the embedding of RDSEFFs into component specifications (see Figure 4.36)

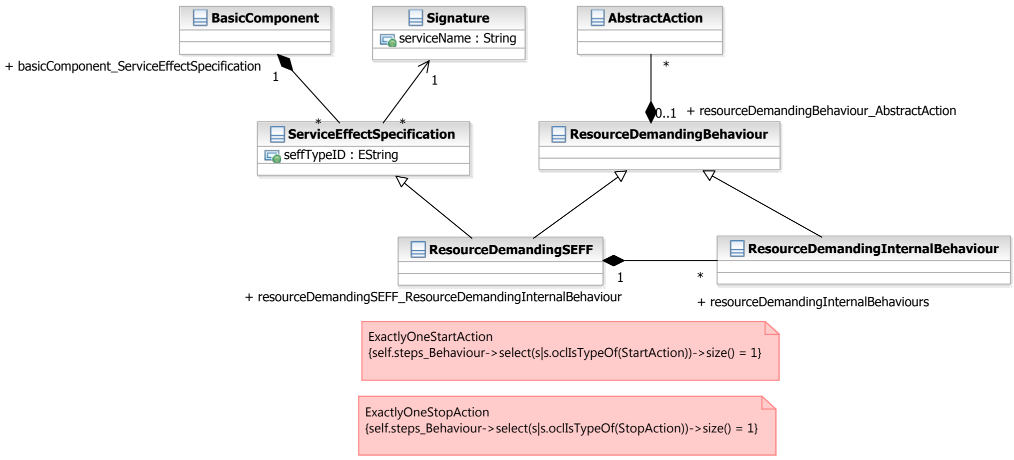


Figure 4.36: BehaviourOverview

Figure Fork Description Overview on fork behaviour elements (see Figure 4.37)

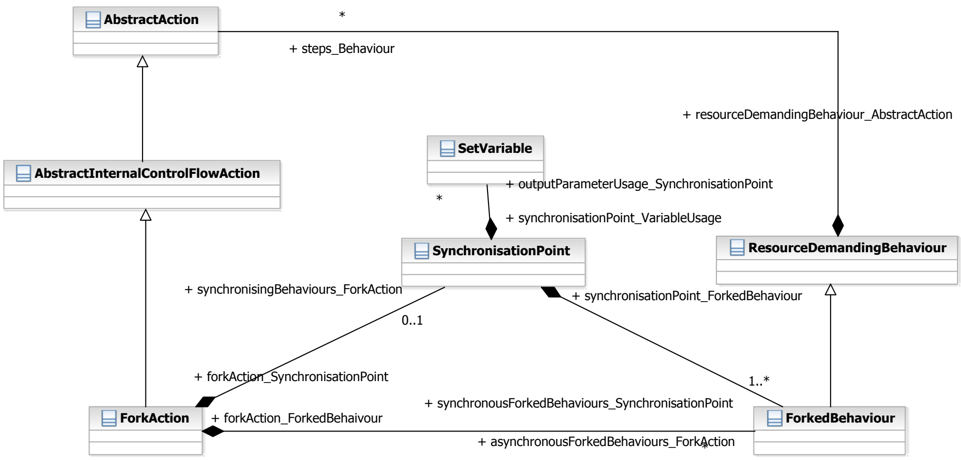


Figure 4.37: Fork

Figure ExternalCall Description Overview on the ExternalCallAction (see Figure 4.38)

Figure PassiveResources Description Overview on passive resources and actions (see Figure 4.39)

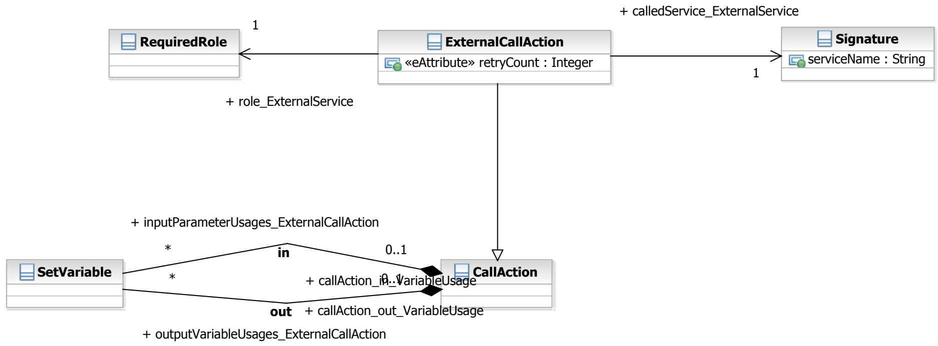


Figure 4.38: ExternalCall

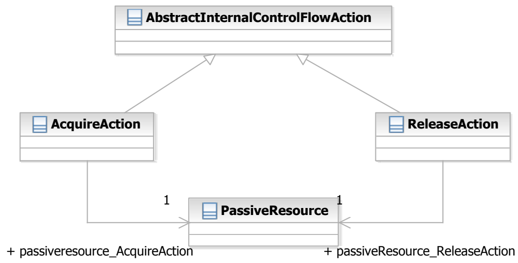


Figure 4.39: PassiveResources

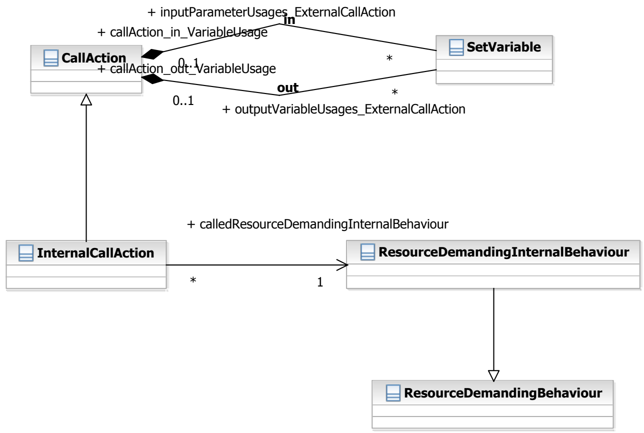


Figure 4.40: InternalCall

Figure InternalCall Description

Figure ActionHierarchy Description Overview on the hierarchy of actions in the PCM SEFF (see Figure 4.41)

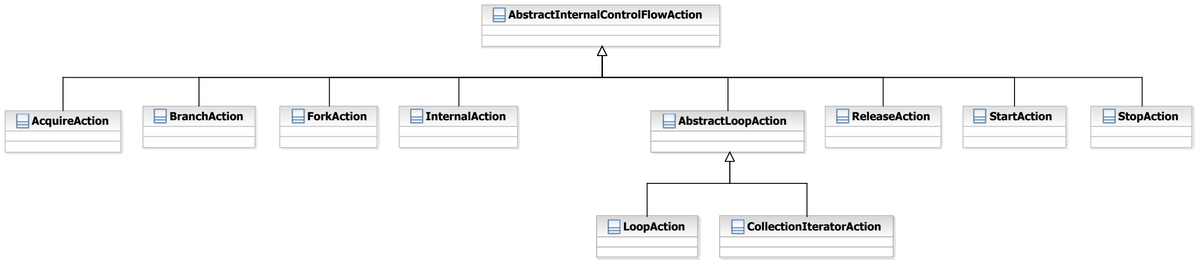


Figure 4.41: ActionHierarchy

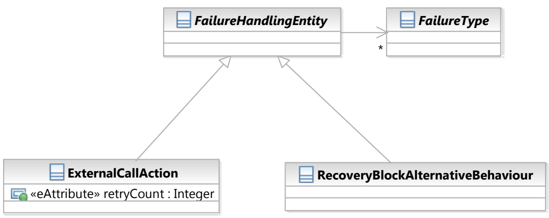


Figure 4.42: FailureHandlingEntity

Figure FailureHandlingEntity Description

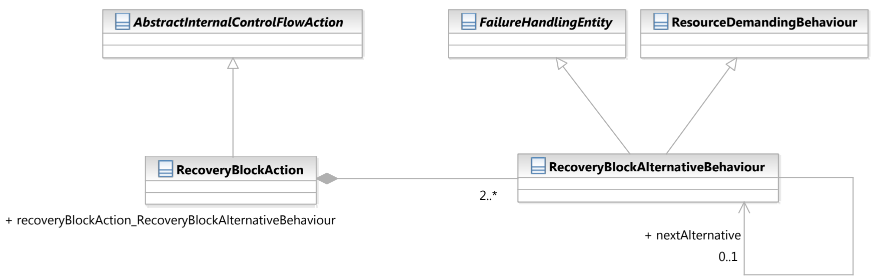


Figure 4.43: RecoveryBlocks

Figure RecoveryBlocks Description

Figure FailureOccurrenceDescription Description

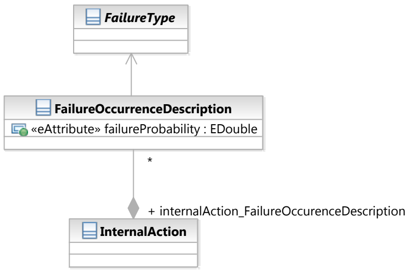


Figure 4.44: FailureOccurrenceDescription

Figure Performance Description Relation between ParametricResourceDemand and AbstractInternalControlFlowAction (see Figure 4.46)

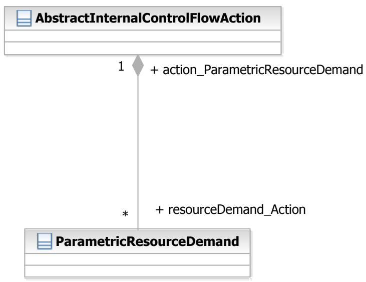


Figure 4.45: Performance

4.16.3 Detailed Class Documentation

4.16.3.1 Class `AbstractAction`

Overview AbstractActions model either a service?s internal computations or calls to external (i.e., required) services, or describe some form of control flow alteration (i.e., branching, loop, or fork). The following first clarifies the notions of internal and external actions, whose meta-classes both inherit from AbstractAction. The RDSEFF defines the control flow between internal and external actions with the predecessor/successor relationship between AbstractActions to model sequential executions. Additionally, special actions for branching, loops, and forks allow other kinds of control flow. Other than flowcharts or UML activity diagrams, the RDSEFF language (as well as the usage model language) requires developers to make the branching, loop, fork bodies explicit using nested ResourceDemandingBehaviours. It disallows backward references in the chain of AbstractActions, which are basically goto statements and can lead to ambiguities and difficult maintainability. For example, this might lead to intertwined control flows as in the example in Fig. 4.9(a), where both the sequences ?abcabcdbcd? and ?abcdbcabcd? could be occur if each backward reference is executed once, which might lead to different execution times. Backward references also allow the specification of loops with multiple entry points as in Fig. 4.9(b). This is not desirable, as the number of loop iterations cannot be specified directly in these cases, which is however necessary for accurate performance prediction. If a developer would specify that each backward link in Fig. 4.9(b) is executed only once, both sequences ?ababc? and ?abcababc? would be possible although they would have different execution times, as ?a? is executed three times in the latter case. To avoid such ambiguities, control flow in the PCM RDSEFF and usage model must be specified without backward references in the chain of AbstractActions. Branches, loops, forks, and their respective bodies have to be made explicit in the specification using nested ResourceDemandingBehaviours.

Class Properties Class `AbstractAction` has the following properties:

predecessor_AbstractAction : AbstractAction [0..1]

resourceDemandingBehaviour_AbstractAction : ResourceDemandingBehaviour [0..1]

successor_AbstractAction : AbstractAction [0..1]

Parent Classes

- Entity (see section 4.7.3.2 on page 97)

4.16.3.2 Class `AbstractBranchTransition`

Overview Two types of branch transitions exist which correspond to the two types of branches. The types cannot be mixed. Either all branch transitions of one BranchAction are probabilistic or guarded.

Class Properties Class `AbstractBranchTransition` has the following properties:

branchAction_AbstractBranchTransition : BranchAction

branchBehaviour_BranchTransition : ResourceDemandingBehaviour

Parent Classes

- Entity (see section 4.7.3.2 on page 97)

4.16.3.3 Class `AbstractInternalControlFlowAction`

Overview Abstract parent class of any internal control flow (e.g. InternalAction)

Class Properties Class `AbstractInternalControlFlowAction` has the following properties:

resourceDemand_Action : ParametricResourceDemand [0..∗]

Parent Classes

- AbstractAction (see section 4.16.3.1 on page 140)

4.16.3.4 Class `AbstractLoopAction`

Overview Abstract parent class of any loop (e.g. LoopAction)

Class Properties Class `AbstractLoopAction` has the following properties:

bodyBehaviour_Loop : ResourceDemandingBehaviour

Parent Classes

- AbstractInternalControlFlowAction (see section 4.16.3.3 on page 141)

4.16.3.5 Class `AcquireAction`

Overview In an RDSEFF, component developers can specify an AcquireAction, which references a passive resource types. Once analysis tools execute this action, they decrease the amount of items available from the referenced passive resource type by one, if at least one item is available. If none item is available, because other, concurrently executed requests have acquired all of them, analysis tools enqueue the current request (first-come first-serve scheduling policy) and block it?s further execution. Acquisition and release of passive resources happen instantaneously and do not consume any time except for waiting delays before actual acquisition. Resource locking may introduce deadlocks when simulating the model, however, for performance analysis with the PCM it is assumed that no deadlocks occur. Otherwise, the model first needs to be fixed accordingly before carrying out the performance prediction.

Class Properties Class `AcquireAction` has the following properties:

passiveresource_AcquireAction : PassiveResource

Parent Classes

- AbstractInternalControlFlowAction (see section 4.16.3.3 on page 141)

4.16.3.6 Class `BranchAction`

Overview The BranchAction splits the RDSEFF control flow with an XOR-semantic, meaning that the control flow continues on exactly one of its attached AbstractBranchTransitions. The RDSEFF supports two different kinds of branch transitions, GuardedBranchTransitions, and ProbabilisticBranchTransitions. RDSEFFs do not allow to use both kinds of transitions on a single BranchAction. Analysis or simulation tools must select exactly one transition based on the included guard or probability, before continuing at a BranchAction.

Class Properties Class `BranchAction` has the following properties:

branches_Branch : AbstractBranchTransition [0..$*$]

Constraints

EitherGuardedBranchesOrProbabilisiticBranchTransitions:

```
self.branches_Branch->forAll(bt|bt.oclIsTypeOf(ProbabilisticBranchTransition))
or self.branches_Branch->forAll(bt|bt.oclIsTypeOf(GuardedBranchTransition))
```

AllProbabilisticBranchProbabilitiesMustSumUpTo1:

```
if self.branches_Branch->forAll(oclIsTypeOf(ProbabilisticBranchTransition)) then
        self.branches_Branch->select(pbt|pbt.oclIsTypeOf(ProbabilisticBranchTransition↘
          →))->collect(pbt|pbt.oclAsType(ProbabilisticBranchTransition).↘
          →branchProbability)->sum() > 0.999
        and self.branches_Branch->select(pbt|pbt.oclIsTypeOf(↘
          →ProbabilisticBranchTransition))->collect(pbt|pbt.oclAsType(↘
          →ProbabilisticBranchTransition).branchProbability)->sum() < 1.001
        else true
endif
```

Parent Classes

- AbstractInternalControlFlowAction (see section 4.16.3.3 on page 141)

4.16.3.7 Class CallAction

Overview Generic class realising call relations between behaviours (e.g. method call).

Class Properties Class CallAction has the following properties:

inputParameterUsages_ExternalCallAction : SetVariable [0..∗]

outputVariableUsages_ExternalCallAction : SetVariable [0..∗]

4.16.3.8 Class CollectionIteratorAction

Overview Collection Iterator Action Models the repeated execution of its inner ResourceDemandingBehaviour for each element of a collection data type. Therefore it contains a reference to an input parameter of the service?s signature, which must be of type CollectionDataType. The NUMBER OF ELEMENTS must be specified from the outside of the component, either by another RDSEFF or by an usage model calling this service. It can be of type integer or IntPMF. Besides the source of the number of iterations, CollectionIteratorActions differ from LoopAction only in their allowed stochastic dependence of random variables inside the loop body?s ResourceDemandingBehaviour. If the same random variable occurs twice in such a loop body, analysis tools must evaluate the second occurrence in stochastic dependence to the first occurrence. This complicates the involved calculation and might lead to the intractability of the model, therefore component developers should use CollectionIteratorActions with care and only include them if they expect that the prediction results would be vastly inaccurate without it.

Parent Classes

- AbstractLoopAction (see section 4.16.3.4 on page 142)

4.16.3.9 Class `ExternalCallAction`

Overview ExternalCallAction models the invocation of a service specified in a required interface. Therefore, it references a Role, from which the providing component can be derived, and a Signature to specify the called service. ExternalCallActions model synchronous calls to required services, i.e., the caller waits until the called service finishes execution before continuing execution itself. The PCM allows modelling asynchronous calls to required services by using an ExternalCallAction inside a ForkedBehaviour. ExternalCallActions do not have resource demands by themselves. Component developers need to specify the resource demand of the called service in the RDSEFF of that service. The resource demand can also be calculated by analysing the providing component. This keeps the RDSEFF specification of different component developers independent from each other and makes them replaceable in an architectural model. ExternalCallActions may contain two sets of VariableUsages specifying input parameter characterisations and output parameter characterisations respectively. VariableUsages for input parameters may only reference IN or INOUT parameters of the call?s referenced signature. The random variable characterisation inside such a VariableUsage may be constants, probability distribution functions, or include a stochastic expression involving for example arithmetic operations. The latter models a dependency between the current service?s own input parameters and the input parameters of the required service.

Class Properties Class `ExternalCallAction` has the following properties:

calledService_ExternalService : Signature

retryCount : Integer
Specifies the number of retries this ExternalCallAction shoul be re-executed in case of failure occurence.

role_ExternalService : RequiredRole

Constraints

SignatureBelongsToRole:

```
<pre id="comment_text_0" class="bz_comment_text">
```

```
check if the signature
(declared in calledService_ExternalService attribute) belongs to the role
(declared in role_ExternalService attribute)
</pre>
```

Parent Classes

- AbstractAction (see section 4.16.3.1 on page 140) ,
- CallAction (see section 4.16.3.7 on page 143) ,
- FailureHandlingEntity (see section 4.16.3.10 on page 145)

4.16.3.10 Class `FailureHandlingEntity`

Overview Failure handling entities are any program constructs that can handle failures. Instances of failure handling entities specify any number of failure types that can be handled.

Class Properties Class `FailureHandlingEntity` has the following properties:

failuretype : FailureType [0..*]

4.16.3.11 Class `FailureOccurrenceDescription`

Overview Describes the occurrence probability of failures of a specified type. In one InternalAction the sum of all failure probabilities must be less than or equal 1.0. Internal actions may only have one failure occurrence descripton for a failure type. (see constraints)

Class Properties Class `FailureOccurrenceDescription` has the following properties:

failureProbability : EDouble

failureType : FailureType

internalAction_FailureOccurenceDescription : InternalAction

4.16.3.12 Class `ForkAction`

Overview Fork Action Splits the RDSEFF control flow with an AND-semantic, meaning that it invokes several ForkedBehaviours concurrently. ForkActions allow both asynchronously and synchronously forked behaviours. Synchronously ForkedBehaviours execute concurrently and the control flow waits for each of these behaviours to terminate before continuing. Each ForkedBehaviour can be considered as a program thread. All parameter characterisations from the surrounding RDSEFF are also valid inside the ForkedBehaviours and can be used to parameterise resource demands or control flow constructs. The parameter characterisations are the same in each ForkedBehaviour. Component developers can use a SynchronisationPoint to join synchronously ForkedBehaviours and specify a result of the computations with its attached VariableUsages. Asynchronously ForkedBehaviours also execute concurrently, but the control flow does not wait for them to terminate and continues immediately after their invocation with the successor action of the ForkAction. Therefore, there is no need for a SynchronisationPoint in this case. It is furthermore not possible to refer to results or output parameters of asynchronously ForkedBehaviours in the rest of the RDSEFF, as it is unclear when these results will be available. The same Fork Action can contain asynchronous and synchronousForkedbehaviours at the same time. In that case, all forked behaviours are started. The control flow waits for all synchronous behaviours to finish execution and only then continues.

Class Properties Class `ForkAction` has the following properties:

asynchronousForkedBehaviours_ForkAction : ForkedBehaviour [0..∗]

synchronisingBehaviours_ForkAction : SynchronisationPoint [0..1]

Parent Classes

- AbstractInternalControlFlowAction (see section 4.16.3.3 on page 141)

4.16.3.13 Class `ForkedBehaviour`

Overview A ForkedBehaviour can be considered as a program thread. All parameter characterisations from the surrounding RDSEFF are also valid inside the ForkedBehaviours and can be used to parameterise resource demands or control flow constructs. The parameter characterisations are the same in each ForkedBehaviour.

Class Properties Class `ForkedBehaviour` has the following properties:

forkAction_ForkedBehaivour : ForkAction [0..1]

synchronisationPoint_ForkedBehaviour : SynchronisationPoint [0..1]

Parent Classes

- ResourceDemandingBehaviour (see section 4.16.3.22 on page 151)

4.16.3.14 Class `GuardedBranchTransition`

Overview Guarded Branch Transition Provides a link between a BranchAction and a nested ResourceDemandingBehaviour, which includes the actions executed inside the branch. It uses a guard, i.e. a boolean expression specified by a RandomVariable, to determine whether the transition is chosen. If the guard evaluates to true, the branch is chosen, otherwise if the guard evaluates to false another branch transition must be chosen. The guard may contain references to the service?s input parameters or component parameters. A component developer can specify complex boolean expressions by using the AND, OR, and NOT operations provided by the StoEx framework. As the domain expert may have characterised the parameters used in a guard with probability distributions, it might happen that a guard does not evaluate to true or false with a probability of 1.0. For example, the specification can express that a guard evaluates to true with a probability of 0.3, and to false with a probability of 0.7. In any case, the probabilities of the individual guards attached to all GuardedBranchTransitions contained in a BranchAction must sum up to 1.0. There is no predefined order in evaluating the guards attached to a BranchAction. This differs from programming languages such as C or Java, where the conditions on if/then/else statements are evaluated in the order of their appearance in the code. Such programming languages allow overlapping branching conditions (for example, if (X <10) //... else if (X <20) // ...), which are not allowed for the guards in GuardedBranchTransitions, because the missing order specification would lead to ambiguous boolean expressions and enable more than one guard to become true. If X would have the value 5, both conditions would evaluate to true if they would be used directly as guards in GuardedBranchTransitions. The correct specification of the guards in this case would be X.VALUE 10 and X.VALUE 10 AND X.VALUE 20. Guards might lead to stochastic dependencies when evaluating variable characterisations inside a branched behaviour. For example, if the guard X.VALUE 10 had formerly evaluated to true, and the RDSEFF uses X.VALUE inside the branched behaviour, the sample space of the random variable specifying the characterisation must be restricted, as the event that X takes a values greater than 10 cannot occur anymore. Therefore its probability is zero. Any variable characterisation always needs to be evaluated under the condition that all guards in the usage scenario?s path to it have evaluated to true.

Class Properties Class `GuardedBranchTransition` has the following properties:

branchCondition_GuardedBranchTransition : PCMRandomVariable

Parent Classes

- AbstractBranchTransition (see section 4.16.3.2 on page 141)

4.16.3.15 Class `InternalAction`

Overview Internal Action Combines the execution of a number of internal computations by a component service in a single model entity. It models calculations inside a component service, which do not include calls to required services. For a desired high abstraction level, an RDSEFF has only one InternalAction for all instructions between two calls to required services. A high abstraction level is needed to keep the model tractable for mathematical analysis methods. However, in principle it is also possible to use multiple InternalActions in direct succession to model on a lower abstraction level and enable more accurate predictions. InternalActions provide an abstraction from the complete behaviour (i.e., control and data flow) of a component service, as they can hide different possible control and data flows not affecting external service calls and express their resource demands as a single stochastic expression. This abstraction underlies the assumption that the resource demands of a number of instruction can be captured sufficiently accurate enough in one such expression

Class Properties Class `InternalAction` has the following properties:

failureOccurrenceDescriptions : FailureOccurrenceDescription [0..∗]

Constraints

Multiple usages of same failure type are not allowed:

```
self.failureOccurrenceDescriptions->forAll(x : FailureOccurrenceDescription,y:↘
  →FailureOccurrenceDescription  | x<>y implies x.failureType <> y.failureType )
```

Sum of failure occurrence probabilities must not exceed 1.0:

```
self.failureOccurrenceDescriptions.failureProbability.oclAsType(Real)->sum()<=1
```

Parent Classes

- AbstractInternalControlFlowAction (see section 4.16.3.3 on page 141)

4.16.3.16 Class `InternalCallAction`

Overview A "SubSEFF"-Action: Realises an internal method call within a SEFF.

Class Properties Class `InternalCallAction` has the following properties:

calledResourceDemandingInternalBehaviour : ResourceDemandingInternalBehaviour

Parent Classes

- CallAction (see section 4.16.3.7 on page 143) ,
- AbstractInternalControlFlowAction (see section 4.16.3.3 on page 141)

4.16.3.17 Class `LoopAction`

Overview Models the repeated execution of its inner ResourceDemandingBehaviour for the loop body. The number of repetitions is specified by a random variable evaluating to integer or an IntPMF. The number of iterations specified by the random variable always needs to be bounded, i.e., the probabilities in an IntPMF for iteration numbers above a certain threshold must be zero. Otherwise, it would be possible that certain requests do not terminate, which would complicate performance analyses. The stochastic expression defining the iteration random variable may include references to input or component parameters to model dependencies between the usage profile and the number of loop iterations. Notice, that loop actions should only be modelled if the loop body contains either external service calls or resource demands directed at special resources. Otherwise, control flow loops in component behaviour should be abstracted by subsuming them in InternalAction, which combine a number of instructions. The influence of different iterations length of such internal loops need to be reflected stochastically by the random variable specifying the ParametricResource-Demand of that InternalAction. Other than Markov chains, RDSEFFs do not specify control flow loops with an reentrance and exit probability on each iteration. Such a specification binds the number of loop iterations to a geometrical distribution, which reflects reality only in very seldom cases. But in many practical cases, the number of iterations is a constant, or the probability for higher iteration numbers is higher than for lower ones. This cannot be expressed directly via a Markov chain (also see [DG00]). Inside the ResourceDemandingBehaviour of LoopActions, it is assumed that random variables are stochastically independent. This is not true in reality, and for example leads to wrong predictions if the same random variable is used twice in succession inside a loop body. In this case, the second occurrence is stochastically dependent to the first occurrence, as the value does not change between two occurrences. Therefore, component developers should be aware of such inaccuracies when using random variables twice inside the body behaviour of a LoopAction.

Class Properties Class `LoopAction` has the following properties:

iterationCount_LoopAction : PCMRandomVariable

Parent Classes

- AbstractLoopAction (see section 4.16.3.4 on page 142)

4.16.3.18 Class `ProbabilisticBranchTransition`

Overview a GuardedBranchTransition, this transition provides a link between a BranchAction and a nested ResourceDemandingBehaviour, which includes the actions executed inside the branch. But instead of using a guard, it specifies a branching probability without parameter dependencies. Analysis tools may directly use it to determine the transition where the control flow continues. The probabilities of all ProbabilisticBranchTransitions belonging to a single BranchAction must sum up to 1.0. Although a probabilistic choice at a branch usually does not happen in a computer program, ProbabilisticBranchTransitions provide a convenient way of modelling in case the actual parameter dependency is too hard to determine or too complex to integrate into a guard. It can also be useful for newly designed components, where the parameter dependency on the control flow guard is still be unknown. However, this construct potentially introduces inaccuracies into the performance model, because it does not reflect the influence of input parameters. Therefore, predictions based on this model can be misleading, if the used input parameters would result in different branching probabilities. The component developer cannot foresee this, when specifying the RDSEFF using ProbabilisticBranchTransitions.

Class Properties Class `ProbabilisticBranchTransition` has the following properties:

branchProbability : EDouble

Parent Classes

- AbstractBranchTransition (see section 4.16.3.2 on page 141)

4.16.3.19 Class RecoveryBlockAction

Overview Recover block actions are a generic failure handling technique. A recovery block consists of a a primary algorithm and one or more alternatives that can be used in case of failure. If the primary algorithm fails, the next alternative is chosen. Here the alternatives also support failure types. Alternatives may specify which kind of failures they can handle.

Class Properties Class `RecoveryBlockAction` has the following properties:

recoveryBlockalternativeBehaviours : RecoveryBlockAlternativeBehaviour [2..*]

Constraints

Alternatives form a chain:

```
self.recoveryBlockalternativeBehaviours->isUnique(s: RecoveryBlockAlternativeBehaviour↘
  → | s.nextAlternative) and
self.recoveryBlockalternativeBehaviours->forAll(x:RecoveryBlockAlternativeBehaviour| x↘
  → <> x.nextAlternative)
```

Parent Classes

- AbstractInternalControlFlowAction (see section 4.16.3.3 on page 141)

4.16.3.20 Class RecoveryBlockAlternativeBehaviour

Overview Recovery block alternative haviours represent alternatives of recovery blocks. They are resource demanding behaviours, thus any behaviour can be defined as an alternative.

The alternatives of a recovery block form a chain. They are failure handling entities, i.e. they can handle failures that occur in previous alternatives. If one alternative fails, the next alternative is executed that can handle the failure type.

Class Properties Class `RecoveryBlockAlternativeBehaviour` has the following properties:

nextAlternative : RecoveryBlockAlternativeBehaviour [0..1]

recoveryBlockAction_RecoveryBlockAlternativeBehaviour : RecoveryBlockAction

Parent Classes

- FailureHandlingEntity (see section 4.16.3.10 on page 145) ,
- ResourceDemandingBehaviour (see section 4.16.3.22 on page 151)

4.16.3.21 Class `ReleaseAction`

Overview The ReleaseAction increases the number of available item for the given passive resource type, before the current request can continue. It should be to execute by one of the other concurrent requests. Acquisition and release of passive resources happen instantaneously and do not consume any time except for waiting delays before actual acquisition. Resource locking may introduce deadlocks when simulating the model, however, for performance analysis with the PCM it is assumed that no deadlocks occur. Otherwise, the model first needs to be fixed accordingly before carrying out the performance prediction.

Class Properties Class `ReleaseAction` has the following properties:

passiveResource_ReleaseAction : PassiveResource

Parent Classes

- AbstractInternalControlFlowAction (see section 4.16.3.3 on page 141)

4.16.3.22 Class `ResourceDemandingBehaviour`

Overview Models the behaviour of a component service as a sequence of internal actions with resource demands, control flow constructs, and external calls. Therefore, the class contains a chain of AbstractActions. The emphasis in this type of behaviour is on the resource demands attached to internal actions, which mainly influence performance analysis. Each action in a ResourceDemandingBehaviour references a predecessor and a successor action. Exceptions are the first and last action, which do not reference a predecessor and a successor respectively. A behaviour is valid, if there is a continuous path from the first to last action, which includes all actions. The chain must not include cycles. To specify control flow branches, loops, or forks, component developers need to use special types of actions, which contain nested inner ResourceDemandingBehaviours to specify the behaviour inside branches or loop bodies. Any ResourceDemandingBehaviour can have at most one starting and one finishing action.

Class Properties Class `ResourceDemandingBehaviour` has the following properties:

abstractBranchTransition_ResourceDemandingBehaviour : AbstractBranchTransition [0..1]

abstractLoopAction_ResourceDemandingBehaviour : AbstractLoopAction [0..1]

steps_Behaviour : AbstractAction [0..∗]

Constraints

ExactlyOneStopAction:

```
self.steps_Behaviour->select(s|s.oclIsTypeOf(StopAction))->size() = 1
```

ExactlyOneStartAction:

```
self.steps_Behaviour->select(s|s.oclIsTypeOf(StartAction))->size() = 1
```

EachActionExceptStartActionandStopActionMustHhaveAPredecessorAndSuccessor:

```
not self.steps_Behaviour->select(s|not s.oclIsTypeOf(StartAction) and not s.↘
  →oclIsTypeOf(StopAction))->exists(a|a.oclAsType(AbstractAction).↘
  →predecessor_AbstractAction.oclIsUndefined()) and not self.steps_Behaviour->select(↘
  →s|not s.oclIsTypeOf(StartAction) and not s.oclIsTypeOf(StopAction))->exists(a|a.↘
  →oclAsType(AbstractAction).successor_AbstractAction.oclIsUndefined())
```

4.16.3.23 Class ResourceDemandingInternalBehaviour

Overview Class representing component-internal behaviour not accessible from the component's interface. Comparable to internal method in object-oriented programming. This behaviour can be called from within a resource demanding behaviour using an InternalCallAction.

Class Properties Class ResourceDemandingInternalBehaviour has the following properties:

resourceDemandingSEFF_ResourceDemandingInternalBehaviour : ResourceDemandingSEFF

Parent Classes

- ResourceDemandingBehaviour (see section 4.16.3.22 on page 151)

4.16.3.24 Class ResourceDemandingSEFF

Overview A resource demanding service effect specification (RDSEFF) is a special type of SEFF designed for performance and reliability predictions. Besides dependencies between provided and required services of a component, it additionally includes notions of resource usage, data flow, and parametric dependencies for more accurate predictions. Its control flow is hierarchically structured and can be enhanced with transition probabilities on branches and numbers of iterations on loops. A ResourceDemandingSEFF is a ServiceEffectSpecification and a Resource-DemandingBehaviour at the same time inheriting from both classes. The reason for this construct lies in the fact, that ResourceDemandingBehaviours can be used recursively inside themselves to describe loop bodies or branched behaviours (explained later), and these inner behaviours should not be RDSEFFs themselves

Class Properties Class ResourceDemandingSEFF has the following properties:

resourceDemandingInternalBehaviours : ResourceDemandingInternalBehaviour [0..*]

Parent Classes

- ServiceEffectSpecification (see section 4.16.3.25 on page 153) ,
- ResourceDemandingBehaviour (see section 4.16.3.22 on page 151)

4.16.3.25 Class ServiceEffectSpecification

Overview Service Effect Specification Models the effect of invoking a specific service of a basic component. Therefore, it references a Signature from an Interface, for which the component takes a ProvidedRole, to identify the described service. This class is abstract and SEFFs for specific analysis purposes need to inherit from this class. A BasicComponent may have an arbitrary number of SEFFs. It can have multiple SEFFs of a different type for a single provided service. For example, one SEFF can express all external service calls with no particular order, while another one includes a restricted order, or still another one expresses resource demands of the service. While different SEFF types have been proposed, the only type currently included in the meta-model is the ResourceDemandingSEFF for performance prediction. Different types of SEFFs should not contradict each other if the languages are equally powerful. For example, the order of allowed external service calls should be the same for each SEFF type modelling sequences of such calls if the modelling languages have the same expressiveness. SEFFs are part of a component and not part of an interface, because they are implementation dependent. The SEFFs of a CompositeComponent are not represented in the meta-model and can be derived automatically by connecting the SEFFs of the encapsulated components of its nested AssemblyContexts. Different SEFFs of a single component access the same component parameter specifications. That means that parameter dependencies to the same component parameters in different SEFF types refer also to the same characterisations.

Class Properties Class `ServiceEffectSpecification` has the following properties:

basicComponent_ServiceEffectSpecification : BasicComponent

describedService__SEFF : Signature

seffTypeID : EString

4.16.3.26 Class SetVariableAction

Overview Set Variable Action Assigns a variable characterisation to an OUT parameter, INOUT parameter, or return value of the service. It ensures that performance-relevant output parameter characterisations of a component service are specified to use them to parameterise the calling RDSEFF. A SetVariableAction must only use output parameters on the left hand side of the assignment and must not use input parameter or local variable names, because input parameters cannot be returned and local names should not be exposed to adhere the black box principle. The action is only intended to allow proper data flow modelling (i.e., output parameter passing) between different component services, but not to reveal additional internals of the service the current RDSEFF models. Thus, the assigned characterisation is not accessible in subsequent actions of the current RDSEFF. Notice, that the stochastic expression used in this assignment must characterise the result of the whole computation of the current service. For non-trivial components, this requires a substantial stochastic approximation based on manual abstraction. However, recall that not the actual result of a component service needs to be specified, but only its performance-relevant attributes. For example, to model the return value of a component service compressing a file, using its file size divided by the compression factor as the stochastic expression is usually sufficient, while the value of the compressed file is not of interest in a performance model. Multiple SetVariableActions assigning to the same output parameter might occur at different locations of the control flow in an RDSEFF. In the case of sequences, loops, and fork, the last assignment overwrites the former assignments and gets transferred back to the calling RDSEFF. Therefore, analysis tools may ignore the former assignments. In the case of using a SetVariableAction in two different branches of a BranchAction, only the assignment in the chosen branch is valid and gets transferred back to the caller.

Class Properties Class `SetVariableAction` has the following properties:

localVariableUsages_SetVariableAction : SetVariable [0..*]

Parent Classes

- AbstractInternalControlFlowAction (see section 4.16.3.3 on page 141)

4.16.3.27 Class `StartAction`

Overview StartActions initiate a scenario behaviour and contain only a successor.

Constraints

StartActionPredecessorMustNotBeDefined:

```
self.predecessor_AbstractAction.oclIsUndefined()
```

Parent Classes

- AbstractInternalControlFlowAction (see section 4.16.3.3 on page 141)

4.16.3.28 Class `StopAction`

Overview StopActions end a scenario behaviour and contain only a predecessor.

Constraints

StopActionSuccessorMustNotBeDefined:

```
self.successor_AbstractAction.oclIsUndefined()
```

Parent Classes

- AbstractInternalControlFlowAction (see section 4.16.3.3 on page 141)

4.16.3.29 Class SynchronisationPoint

Overview Component developers can use a SynchronisationPoint to join synchronously ForkedBehaviours and specify a result of the computations with its attached VariableUsages.

Class Properties Class SynchronisationPoint has the following properties:

forkAction_SynchronisationPoint : ForkAction

outputParameterUsage_SynchronisationPoint : SetVariable [0..*]

synchronousForkedBehaviours_SynchronisationPoint : ForkedBehaviour [1..*]

4.17 Package pcm::seff::performance

4.17.1 Package Overview

Package capturing performance aspects of an RDSEFF

4.17.2 Package Diagrams

Figure Performance Description Relation between ParametricResourceDemand and AbstractInternalControlFlowAction (see Figure 4.46)

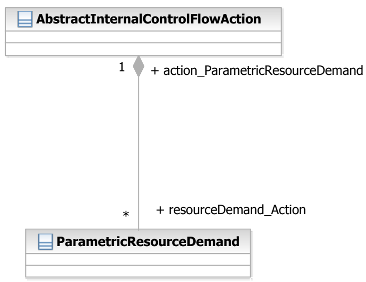


Figure 4.46: Performance

4.17.3 Detailed Class Documentation

4.17.3.1 Class ParametricResourceDemand

Overview Parametric Resource Demand Specifies the amount of processing requested from a certain type of resource in a parametrised way. It assigns the demand specified as a Random-Variable to an abstract ProcessingResourceType (e.g., CPU, hard disk) instead of a concrete ProcessingResourceSpecification (e.g., 5 Ghz CPU, 20 MByte/s hard disk). This keeps the RDSEFF independent from a specific resource environment, and makes the concrete resources replaceable to answer sizing questions. The demand?s unit is equal for all ProcessingResourceSpecifications referencing the same ProcessingResourceType. It can for example be ?WorkUnits? for CPUs [Smi02] or ?BytesRead? for hard disks. Each ProcessingResource- Specification contains a processing rate for demands (e.g., 1000 WorkUnits/s, 20 MB/s), which analysis tools use to compute an actual timing value in seconds. They use this timing value for example as the service demand on a service center in a queueing network or the firing delay of a transition in a Petri net. As multiple component services might request processing on the same resource, these analytical or simulation models allow determining the waiting delay induced by this contention effect. Besides this parameterisation over different resource environments, Parametric- ResourceDemands also parameterise over the usage profile. For this, the stochastic expression specifying the resource demand can contain references to the service?s input parameters or the component parameters. Upon evaluating the resource demand, analysis tools use the current characterisation of the referenced input or component parameter and substitute the reference with this characterisation in the stochastic expression. Solving the stochastic expression, which can be a function involving arithmetic operators (Chapter 3.3.6), then yields a constant or probability function for the resource demand. As an example for solving the parameterisation over resource environment and usage profile, consider an RDSEFF for a service implementing the bubblesort algorithm. It might include a CPU demand specification of n2?2000WorkUnits derived from complexity theory (n2) and empirical measurements (2000). In this case n refers to the length of the list the algorithm shall sort, which is an input parameter of the service. If the current characterisation of the list?s length is 100 (as the modelled usage profile), analysis tools derive 1002 ? 2000 12000 WorkUnits from the specification, thus resolving the usage profile dependency. If the CPU ProcessingResource-Specification the service?s 126 4.3. Resource Demanding Service Effect Specification component is allocated on then contains a processing rate of 10000WorkUnits/s, analysis tools derive an execution time of 12000 WorkUnits 10000 WorkUnits/s = 1:2 s from the specification, thus resolving the resource environment dependency. The stochastic expression for a ParametricResourceDemand depends on the implementation of the service. Component developers can specify it using complexity theory, estimations, or measurements. However, how to get data to define such expressions accurately is beyond of the scope of this thesis. Woodside et al. [WVCB01] and Krogmann [Kro07] present approaches for measuring resource demands in dependency to input parameters. Meyerhoefer et al. [ML05] and Kuperberg et al. [KB07] propose methods to establish resource demands independent from concrete resources. For the scope of this thesis, it is assumed that these methods have been applied and an accurate specification of the ParametricResourceDemand is available.

Class Properties Class `ParametricResourceDemand` has the following properties:

action_ParametricResourceDemand : AbstractInternalControlFlowAction

requiredResource_ParametricResourceDemand : ProcessingResourceType

specification_ParametericResourceDemand : PCMRandomVariable

4.18 Package pcm::subsystem

4.18.1 Package Overview

Package capturing the subsystem entity

4.18.2 Package Diagrams

Figure SubSystem Description A SubSystem is a ComposedProvidingRequiringEntity and a RepositoryComponent (see Figure 4.47)

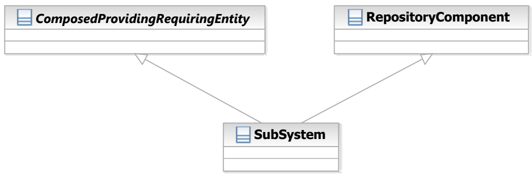


Figure 4.47: SubSystem

4.18.3 Detailed Class Documentation

4.18.3.1 Class SubSystem

Overview A SubSystem is structually comparable to a CompositeComponent. The major difference is the white-blox property it preserves for System Deployers. While Component Developer have a white-box view for their CompositeComponents, a System Deployer perceives a CompositeComponent like any other component as a black-box entity, which thus cannot be allocated onto different nodes in the resource environment (a CompositeComponent cannot be split up at allocation time). Opposed to that, SubSystems are white-box entities for System Deployers, meaning that they can be allocated to different nodes of the resource environment, if required. They are pure logical groupings of components, which can be reused by Component Developers and System Architects like usual components.

Remark 1: If a SubSystem is part of a CompositeComponent (inner component) is looses its white-box property, as there is a outer black-box component hiding the its and consequently the SubSytem's internals.

Remark 2: Structurally, SubSytem can be converted into CompositeComponents and vice versa.

Example: To model a layered architecture, of which each layer is potentially split up to run on multiple machines (in the resource environment), each layer can be represented by a subsystem, allowing to allocated each layer's components individually.

Parent Classes

- ComposedProvidingRequiringEntity (see section 4.7.3.1 on page 96) ,
- RepositoryComponent (see section 4.13.3.20 on page 126)

4.19 Package `pcm::system`

4.19.1 Package Overview

The system package holds only the System meta class. A system is the most high-level and out-most compositional entity of the PCM. It defines the boundaries of a modelled application. Only systems (more precisely provided services of a system) can be accessed from usage profile. Systems also can carry QoS-Annotations, a special means to express fixed QoS properties of services that are required at the system boundary.

4.19.2 Package Diagrams

Figure System Internals Description Overview on the System and its relation to allocation. Exactly one system is subject to allocation. (see Figure 4.48)

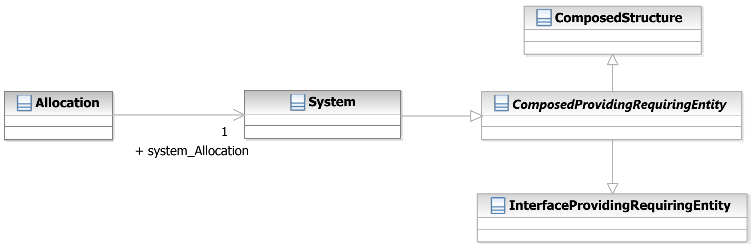


Figure 4.48: System Internals

4.19.3 Detailed Class Documentation

4.19.3.1 Class `System`

Overview A System is the out-most entity of a PCM's assembly of components. It captures the modeling decision to which extend a system under investigation is modelled within the PCM. A System is not composable (part of another composition) because it has QoS annotations which are only allowed in the "outer" composition thing. And a UsageModel must only access ProvidedRoles of a System and not of inner components. The System is inheriting from ComposedProvidingRequiringEntity to have a unique means for expressing the inner composition of an entity (here: System). Also it allows using the same editor etc. being applied to ComposedProvidingRequiringEntity and thus also being useful for System.

A system consists of an assembly and is itself referenced by an allocation (only a System can be allocated). The first specifies how the components are connected with other components, the latter specifies how the components and connectors are mapped to the resource environment (hardware and middleware). From a structural point of view, Systems can be seen as special kind of CompositeComponents. Systems are not supposed to be reused as components are. The are assumed to be coupled by using special techniques for system integration.

A System has provided and required roles like a composite component. Only a system's provided role can be accessed from the usage profile. Only a system's required roles can have QOS annotations.

Like SubSystems, Systems are white-box entities for the Software Architect and also for the System Deployer. Thus, a System Deployer can and must allocate inner components of a System individually.

Class Properties Class `System` has the following properties:

qosAnnotations_System : QoSAnnotations [0..∗]
QoS Annotations allow for specifing fixed (non-parameterised) QoS properties at the system boundary level for required services. For example, for a required service it can be specified that its response time is fixed "3 ms". See the QoS Annotations package for more details.

Constraints

SystemMustHaveAtLeastOneProvidedRole:

```
not self.providedRoles_InterfaceProvidingEntity->isEmpty()
```

Parent Classes

- Entity (see section 4.7.3.2 on page 97) ,
- ComposedProvidingRequiringEntity (see section 4.7.3.1 on page 96)

4.20 Package `pcm::usagemodel`

4.20.1 Package Overview

The usage of a software system by external clients has to be captured in models to enable model-driven performance predictions. Here, the term usage refers to workload (i.e., the number of users concurrently present in the system), usage scenarios (i.e., possible sequences of invoking services at system provided roles), waiting delays between service invocations, and values for parameters and component configurations.

This package contains the usage specification language, which (i) provides more expressiveness for characterising parameter instances than previous models, but (ii) at the same time is restricted to concepts familiar to domain experts to create a domain specific language. The language is called PCM usage model.

The UsageModel specifies the whole user interaction with a system from a performance viewpoint. It consists of a number of concurrently executed UsageScenarios and a set of global UserData specifications. Each UsageScenario includes a workload and a scenario behaviour. The concepts are explained for the single meta classes included in this package.

Note that UsageModels are completely decoupled from the inner contents of a system, which consists of an assembly and a connected resource environment. The UsageModel only refers to services of system provided roles. It regards the component architecture (i.e., the assembly) as well as used resources (i.e., hardware devices such as CPUs and harddisks or software entities such as threads, semaphores) as hidden in the system. Thus, the UsageModel only captures information that is available to domain experts and can be changed by them. Resource environment and component architecture may be changed independently from the UsageModel by system architects, if the system's ProvidedRoles remain unchanged.

Discussion: Notice, that unlike other behavioural description languages for performance prediction (e.g., [162, 187, 78]), the PCM usage model specifically models user behaviour and for example does not refer to resources. Other performance meta-models mix up the specification of user behaviour, component behaviour, and resources, so that a single developer role (i.e., a performance analyst) needs to specify the performance model. Opposed to this, the PCM targets a division of work for multiple developer roles (cf. Section 3.1 of Heiko Koziolek's dissertation).

Furthermore, none of the other performance meta-models support explicit service parameter modelling. While CSM [162] includes a meta-class Message to specify the amount of data transferred between two steps in the performance model, and KLAPER [78] allows the specification of parameter values in principle, none of these language uses the information to parameterise resource demands or component behaviour. Additionally, they do not provide the information readily analysable by MDSD tools.

The PCM usage model has been designed based on meta models such as the performance domain model of the UML SPT profile [31], the Core Scenario Model (CSM) [162], and KLAPER [78]. It is furthermore related to usage models used in statistical testing [34]. Although the concepts included in the PCM usage model are quite similar to the modelling capabilities of the UML SPT profile, there are some subtle differences: - The usage model is aligned with the role of the domain expert, and uses only concepts known to this role. It is a domain specific language, whereas the UML SPT performance domain model is a general purpose language that includes information, which is usually spread over multiple developer roles such as the component assembler and the system deployer, so that a domain expert without a technical background could not specify an instance of it. Nevertheless, domain experts should be able to create PCM usage models with appropriate tools independently from other developer roles, because such models only contain concepts known to them. - The number of loop iterations is not bound to a constant value, but can be specified as a random variable. - The control flow constructs are arranged in a hierarchical fashion to enable easy analyses. - Users are restricted to non-concurrent behaviour, as it is assumed, that users only execute the services of a system one at a time. - System service invocations can be enhanced with characterisations of parameters values.

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

[34] James A. Whittaker and Michael G. Thomason, ?A Markov chain model for statistical software testing,? IEEE Transactions on Software Engineering, vol. 20, no. 10, pp. 812?824, Oct. 1994.

[78] V. Grassi, R. Mirandola, and A. Sabetta, ?From design to analysis models: a kernel language for performance and reliability analysis of component-based systems,? in Proc. 5th International Workshop on Software and Performance (WOSP ?05). New York, NY, USA: ACM Press, 2005, pp. 25?36.

[162] D. B. Petriu and M. Woodside, ?A metamodel for generating performance models from UML designs,? in UML 2004 - The Unified Modeling Language. Model Languages and Applications. 7th International Conference, Lisbon, Portugal, October 11-15, 2004, Proceedings, ser. LNCS, T. Baar, A. Strohmeier, A. Moreira, and S. J. Mellor, Eds., vol. 3273. Springer, 2004, pp. 41?53.

[187] C. U. Smith, C. M. Llado, V. Cortellessa, A. D. Marco, and L. G. Williams, ?From UML models to software performance results: an SPE process based on XML interchange formats,? in Proc. 5th international workshop on Software and performance (WOSP?05). New York, NY, USA: ACM Press, 2005, pp. 87?98.

4.20.2 Package Diagrams

Figure Usage Model Description Overview on the UsageModel (see Figure 4.49)

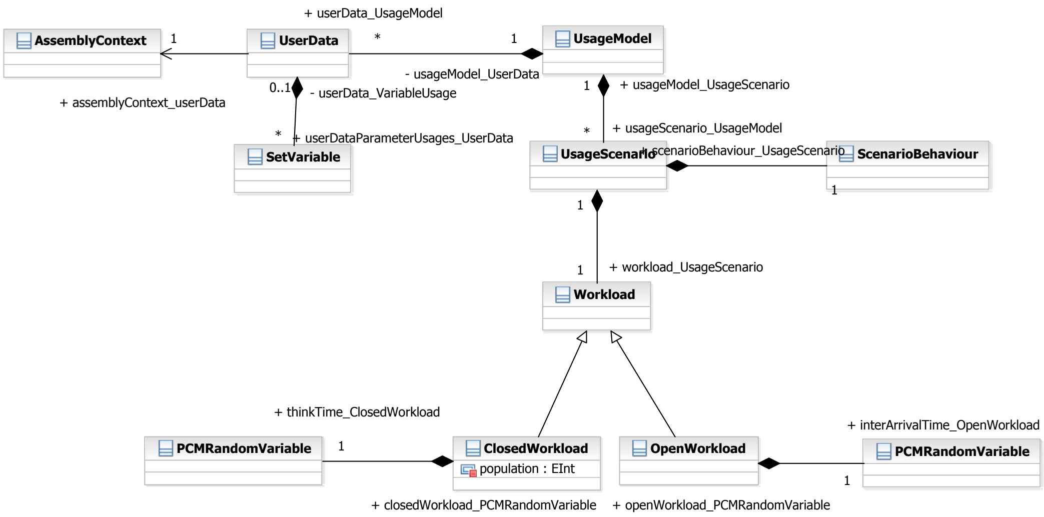


Figure 4.49: Usage Model

Figure EntryLevelSystemCall Description Overview on the EntryLevelSystemCall (see Figure 4.50)

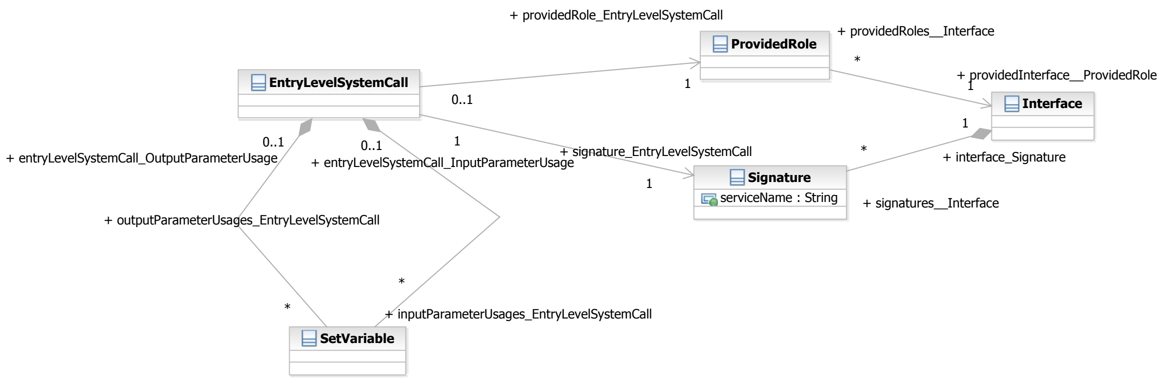


Figure 4.50: EntryLevelSystemCall

Figure UsageModel_UsageScenario_ScenarioBehaviour Description Overview on UsageModel, UsageScenario and Workload (see Figure 4.51)

Figure ScenarioBehaviour Description Overview on ScenarioBehaviour (see Figure 4.52)

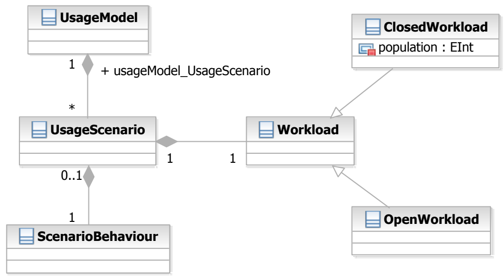


Figure 4.51: UsageModel_UsageScenario_ScenarioBehaviour

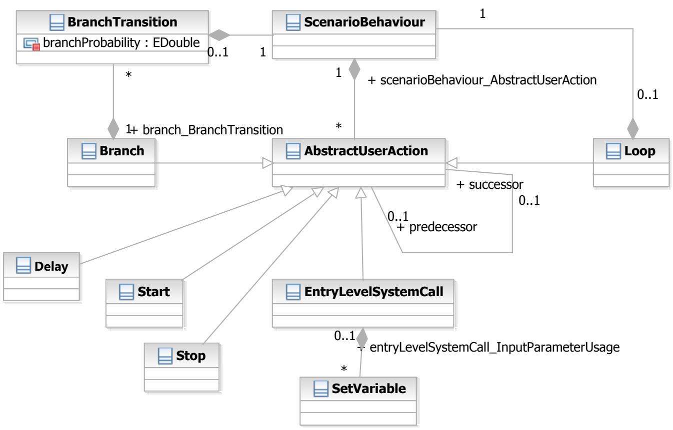


Figure 4.52: ScenarioBehaviour

4.20.3 Detailed Class Documentation

4.20.3.1 Class AbstractUserAction

Overview See the AbstractAction documentation for why it is advantageous to model control flow in this way, as the same principle is used in the RDSEFF language.

Concrete user actions of the usage model are: - Branch - Loop - EntryLevelSystemCall - Delay

Class Properties Class `AbstractUserAction` has the following properties:

predecessor : AbstractUserAction [0..1]

scenarioBehaviour_AbstractUserAction : ScenarioBehaviour

successor : AbstractUserAction [0..1]

Parent Classes

- Entity (see section 4.7.3.2 on page 97)

4.20.3.2 Class Branch

Overview A Branch splits the user flow with a XOR-semantic: one of the included BranchTransitions is taken depending on the specified branch probabilities. Each BranchTransition contains a nested ScenarioBehaviour, which a user executes once this branch transition is chosen. After execution of the complete nested ScenarioBehaviour, the next action in the user flow after the Branch is its successor action.

A constraint ensures that all branchProbabilities of the included BranchTransitions sum up to 1.

Class Properties Class `Branch` has the following properties:

branchTransitions_Branch : BranchTransition [0..∗]

Constraints

AllBranchProbabilitiesMustSumUpTo1:

```
self->collect(branchTransitions_Branch.branchProbability)->sum() > 0.999 and self->
  collect(branchTransitions_Branch.branchProbability)->sum() <1.001
```

Parent Classes

- AbstractUserAction (see section 4.20.3.1 on page 163)

4.20.3.3 Class BranchTransition

Overview The BranchTransition is an association class that realises the containment of ScenarioBehaviours in in the branches of a Branch action. It is a separate meta class because it has the additional attribute branchProbability that specifies how probably it is that the references ScenarioBehaviour is executed in the Branch action.

See also Branch.

Class Properties Class `BranchTransition` has the following properties:

branchProbability : EDouble

branch_BranchTransition : Branch

branchedBehaviour_BranchTransition : ScenarioBehaviour

4.20.3.4 Class ClosedWorkload

Overview ClosedWorkload specifies directly the (constant) user population and a think time. It models that a fixed number of users execute their scenario, then wait (or think) for the specified amount of think time as a RandomVariable, and then reenter the system executing their scenario again. Performance analysts use closed workloads to model scenarios, where the number of users is known (e.g., a fixed number of users in a company).

Class Properties Class `ClosedWorkload` has the following properties:

population : EInt

thinkTime_ClosedWorkload : PCMRandomVariable

Constraints

PopulationInClosedWorkloadNeedsToBeSpecified:

```
not self.population.oclIsUndefined() and self.population <> ''
```

ThinkTimeInClosedWorkloadNeedsToBeSpecified:

```
not self.thinkTime_ClosedWorkload.oclIsUndefined() and self.thinkTime_ClosedWorkload.↘
  →specification <> ''
```

Parent Classes

- Workload (see section 4.20.3.15 on page 170)

4.20.3.5 Class `Delay`

Overview A Delay represents a timing delay as a RandomVariable between two user actions. The Delay is included into the usage model to express that users do not call system services in direct successions, but usually need some time to determine their next action. User delays are for example useful, if a performance analyst wants to determine the execution time for a complete scenario behaviour (instead of a single service), which needs to include user delays.

Class Properties Class `Delay` has the following properties:

timeSpecification_Delay : PCMRandomVariable

Parent Classes

- AbstractUserAction (see section 4.20.3.1 on page 163)

4.20.3.6 Class `EntryLevelSystemCall`

Overview An EntryLevelSystemCall models the call to a service provided by a system. Therefore, an EntryLevelSystemCall references a ProvidedRole of a PCM System, from which the called interface and the providing component within the system can be derived, and a Signature specifying the called service. Notice, that the usage model does not permit the domain expert to model calls directly to components, but only to system roles. (TODO: Add such a constraint.) This decouples the System structure (i.e., the component-based software architecture model and its allocation) from the UsageModel and the software architect can change the System (e.g., include new components, remove existing components, or change their wiring or allocation) independently from the domain expert, if the system provided roles are not affected. EntryLevelSystemCalls may include a set of input parameter characterisations and a set of output parameter characterisations (as described in the pcm::parameters package). However, the random variables characterising the input parameters like NUMBER_OF_ELEMENTS can not depend on other variables in the usage model. They have to be composed from literals only including literals describing random variables having a certain fixed distribution.

Class Properties Class `EntryLevelSystemCall` has the following properties:

inputParameterUsages_EntryLevelSystemCall : SetVariable [0..$*$]

outputParameterUsages_EntryLevelSystemCall : SetVariable [0..$*$]

providedRole_EntryLevelSystemCall : ProvidedRole

signature_EntryLevelSystemCall : Signature

Parent Classes

- AbstractUserAction (see section 4.20.3.1 on page 163)

4.20.3.7 Class Loop

Overview A Loop models a repeated sequence of actions in the user flow. It contains a nested ScenarioBehaviour specifying the loop body, and a RandomVariable specifying the number of iterations.

Class Properties Class Loop has the following properties:

bodyBehaviour_Loop : ScenarioBehaviour

loopIteration_Loop : PCMRandomVariable

Parent Classes

- AbstractUserAction (see section 4.20.3.1 on page 163)

4.20.3.8 Class OpenWorkload

Overview OpenWorkload specifies usage intensity with an inter-arrival time (i.e., the time between two user arrivals at the system) as a RandomVariable with an arbitrary probability distribution. It models that an infinite stream of users arrives at a system. The users execute their scenario, and then leave the system. The user population (i.e., the number of users concurrently present in a system) is not fixed in an OpenWorkload.

Class Properties Class `OpenWorkload` has the following properties:

interArrivalTime_OpenWorkload : PCMRandomVariable

Constraints

InterArrivalTimeInOpenWorkloadNeedsToBeSpecified:

```
not self.interArrivalTime_OpenWorkload.oclIsUndefined() and self.↘
  →interArrivalTime_OpenWorkload.specification <> ''
```

Parent Classes

- Workload (see section 4.20.3.15 on page 170)

4.20.3.9 Class ScenarioBehaviour

Overview A ScenarioBehaviour specifies possible sequences of executing services provided by the system. It contains a set of AbstractUserActions, each referencing a predecessor and successor (except the first and last action), thereby forming a sequence of actions.

See the AbstractAction documentation for why it is advantageous to model control flow in this way, as the same principle is used in the RDSEFF language.

Concrete user actions of the usage model are: - Branch - Loop - EntryLevelSystemCall - Delay - Start - Stop

So far, ScenarioBehaviours do not include forks in the user flow (i.e., splitting the flow with an AND semantic), as it is assumed that users always act sequentially.

As there are no random variables depending on other variables in the usage model, there are no equivalent actions to GuardedBranchTransitions or CollectionIteratorActions.

Class Properties Class `ScenarioBehaviour` has the following properties:

actions_ScenarioBehaviour : AbstractUserAction [0..*]

branchTransition_ScenarioBehaviour : BranchTransition [0..1]

loop_ScenarioBehaviour : Loop [0..1]

usageScenario_SenarioBehaviour : UsageScenario [0..1]

Constraints

Exactly one start:

```
self.actions_ScenarioBehaviour->select(s|s.oclIsTypeOf(Start))->size() = 1
```

Exactly one stop:

```
self.actions_ScenarioBehaviour->select(s|s.oclIsTypeOf(Stop))->size() = 1
```

Each user action except Start and Stop must have a predecessor and successor:

```
not self.actions_ScenarioBehaviour->select(s|not s.oclIsTypeOf(Start) and not s.↘
  →oclIsTypeOf(Stop))->exists(a|a.oclAsType(AbstractUserAction).predecessor.↘
  →oclIsUndefined()) and not self.actions_ScenarioBehaviour->select(s|not s.↘
  →oclIsTypeOf(Start) and not s.oclIsTypeOf(Stop))->exists(a|a.oclAsType(↘
  →AbstractUserAction).successor.oclIsUndefined())
```

Parent Classes

- Entity (see section 4.7.3.2 on page 97)

4.20.3.10 Class `Start`

Overview Each ScenarioBehaviour has exactly one Start action which marks the action where the control flows begins. Start actions have no predecessor.

Constraints

StartHasNoPredecessor:

```
self.predecessor.oclIsUndefined()
```

Parent Classes

- AbstractUserAction (see section 4.20.3.1 on page 163)

4.20.3.11 Class `Stop`

Overview Each ScenarioBehaviour has exactly one Stop action which marks the action where the control flows ends. Stop actions have no successor.

Constraints

StopHasNoSuccessor:

```
self.successor.oclIsUndefined()
```

Parent Classes

- AbstractUserAction (see section 4.20.3.1 on page 163)

4.20.3.12 Class `UsageModel`

Overview The UsageModel specifies the whole user interaction with a system from a performance viewpoint. It consists of a number of concurrently executed UsageScenarios and a set of global UserData specifications. Each UsageScenario includes a workload and a scenario behaviour.

Class Properties Class `UsageModel` has the following properties:

usageScenario_UsageModel : UsageScenario [0..*]

userData_UsageModel : UserData [0..*]

4.20.3.13 Class `UsageScenario`

Overview UsageScenarios are concurrently executed behaviours of users within one UsageModel. It describes which services are directly invoked by users in one specific use case and models the possible sequences of calling them. Each UsageScenario includes a workload and a scenario behaviour.

Class Properties Class `UsageScenario` has the following properties:

scenarioBehaviour_UsageScenario : ScenarioBehaviour

usageModel_UsageScenario : UsageModel

workload_UsageScenario : Workload

Parent Classes

- Entity (see section 4.7.3.2 on page 97)

4.20.3.14 Class `UserData`

Overview UserData characterises data used in specific assembly contexts in the system. This data is the same for all UsageScenarios, i.e., multiple users accessing the same components access the same data. This UserData refers to component parameters of the system publicized by the software architect (see pcm::parameters package). The domain expert characterises the values of component parameters related to business concepts (e.g., user specific data, data specific for a business domain), whereas the software architect characterises the values of component parameters related to technical concepts (e.g., size of caches, size of a thread pool, configuration data, etc.). One UserData instance includes all parameter characterisation for the annotated entity.

Class Properties Class `UserData` has the following properties:

assemblyContext_userData : AssemblyContext

usageModel_UserData : UsageModel

userDataParameterUsages_UserData : SetVariable [0..*]

4.20.3.15 Class Workload

Overview A Workload specifies the usage intensity of a system, which relates to the number of users concurrently present in the system. The PCM usage model adopts this concept from classical queueing theory [123]. The specified workloads can directly be used in queueing networks or easily be mapped to markings in stochastic Petri nets. Workloads can either be open or closed.

The algorithms used to analyse queueing networks differ depending on whether open or closed workloads are modelled [123]. Some special queueing networks can only be analysed given a particular workload type (open or closed). Notice, that it is possible to specify a usage model with open workload usage scenarios and closed workload usage scenarios at the same time. Open and closed workloads can be executed in parallel when analysing the model.

[123] E. Lazowska, J. Zahorjan, G. Graham, and K. Sevcik, Quantitative System Performance. Prentice Hall, 1984.

Class Properties Class `Workload` has the following properties:

usageScenario_Workload : UsageScenario

4.21 Package stoex

4.21.1 Package Overview

4.21.2 Detailed Class Documentation

4.21.2.1 Class AbstractNamedReference

Overview

Class Properties Class `AbstractNamedReference` has the following properties:

referenceName : String

4.21.2.2 Class Atom

Overview

Parent Classes

- Unary (see section 4.21.2.28 on page 177)

4.21.2.3 Class BoolLiteral

Overview

Class Properties Class `BoolLiteral` has the following properties:

value : Boolean

Parent Classes

- Atom (see section 4.21.2.2 on page 170)

4.21.2.4 Class `BooleanExpression`

Overview

Parent Classes

- IfElse (see section 4.21.2.11 on page 172)

4.21.2.5 Class `BooleanOperatorExpression`

Overview

Class Properties Class `BooleanOperatorExpression` has the following properties:

left : BooleanExpression

operation : BooleanOperations

right : BooleanExpression

Parent Classes

- BooleanExpression (see section 4.21.2.4 on page 171)

4.21.2.6 Class `CompareExpression`

Overview

Class Properties Class `CompareExpression` has the following properties:

left : Term

operation : CompareOperations

right : Term

Parent Classes

- Comparison (see section 4.21.2.7 on page 172)

4.21.2.7 Class Comparison

Overview

Parent Classes

- BooleanExpression (see section 4.21.2.4 on page 171)

4.21.2.8 Class DoubleLiteral

Overview

Class Properties Class `DoubleLiteral` has the following properties:

value : EDouble

Parent Classes

- NumericLiteral (see section 4.21.2.17 on page 174)

4.21.2.9 Class Expression

Overview

4.21.2.10 Class FunctionLiteral

Overview

Class Properties Class `FunctionLiteral` has the following properties:

id : String

parameters_FunctionLiteral : Expression [0..∗]

Parent Classes

- Atom (see section 4.21.2.2 on page 170)

4.21.2.11 Class IfElse

Overview

Parent Classes

- Expression (see section 4.21.2.9 on page 172)

4.21.2.12 Class `IfElseExpression`

Overview

Class Properties Class `IfElseExpression` has the following properties:

conditionExpression : BooleanExpression

elseExpression : BooleanExpression

ifExpression : BooleanExpression

Parent Classes

- IfElse (see section 4.21.2.11 on page 172)

4.21.2.13 Class `IntLiteral`

Overview

Class Properties Class `IntLiteral` has the following properties:

value : Integer

Parent Classes

- NumericLiteral (see section 4.21.2.17 on page 174)

4.21.2.14 Class `NamespaceReference`

Overview

Class Properties Class `NamespaceReference` has the following properties:

innerReference_NamespaceReference : AbstractNamedReference

Parent Classes

- AbstractNamedReference (see section 4.21.2.1 on page 170)

4.21.2.15 Class `NegativeExpression`

Overview

Class Properties Class `NegativeExpression` has the following properties:

inner : Unary

Parent Classes

- Unary (see section 4.21.2.28 on page 177)

4.21.2.16 Class `NotExpression`

Overview

Class Properties Class `NotExpression` has the following properties:

inner : Unary

Parent Classes

- Unary (see section 4.21.2.28 on page 177)

4.21.2.17 Class `NumericLiteral`

Overview

Parent Classes

- Atom (see section 4.21.2.2 on page 170)

4.21.2.18 Class `Parenthesis`

Overview

Class Properties Class `Parenthesis` has the following properties:

innerExpression : Expression

Parent Classes

- Atom (see section 4.21.2.2 on page 170)

4.21.2.19 Class `Power`

Overview

Parent Classes

- Product (see section 4.21.2.22 on page 175)

4.21.2.20 Class PowerExpression

Overview

Class Properties Class PowerExpression has the following properties:

base : Power

exponent : Unary

Parent Classes

- Power (see section 4.21.2.19 on page 174)

4.21.2.21 Class ProbabilityFunctionLiteral

Overview

Class Properties Class ProbabilityFunctionLiteral has the following properties:

function_ProbabilityFunctionLiteral : null

Parent Classes

- Atom (see section 4.21.2.2 on page 170)

4.21.2.22 Class Product

Overview

Parent Classes

- Term (see section 4.21.2.26 on page 176)

4.21.2.23 Class ProductExpression

Overview

Class Properties Class ProductExpression has the following properties:

left : Product

operation : ProductOperations

right : Power

Parent Classes

- Product (see section 4.21.2.22 on page 175)

4.21.2.24 Class RandomVariable

Overview

Class Properties Class `RandomVariable` has the following properties:

expression : Expression

specification : String

4.21.2.25 Class StringLiteral

Overview

Class Properties Class `StringLiteral` has the following properties:

value : String

Parent Classes

- Atom (see section 4.21.2.2 on page 170)

4.21.2.26 Class Term

Overview

Parent Classes

- Comparison (see section 4.21.2.7 on page 172)

4.21.2.27 Class TermExpression

Overview

Class Properties Class `TermExpression` has the following properties:

left : Term

operation : TermOperations

right : Product

Parent Classes

- Term (see section 4.21.2.26 on page 176)

4.21.2.28 Class Unary

Overview

Parent Classes

- Power (see section 4.21.2.19 on page 174)

4.21.2.29 Class Variable

Overview

Class Properties Class Variable has the following properties:

id_Variable : AbstractNamedReference

Parent Classes

- Atom (see section 4.21.2.2 on page 170)

4.21.2.30 Class VariableReference

Overview

Parent Classes

- AbstractNamedReference (see section 4.21.2.1 on page 170)

4.22 Package probfunction

4.22.1 Package Overview

4.22.2 Detailed Class Documentation

4.22.2.1 Class BoxedPDF

Overview

Class Properties Class BoxedPDF has the following properties:

samples : ContinuousSample [0..∗]

Parent Classes

- ProbabilityDensityFunction (see section 4.22.2.9 on page 179)

4.22.2.2 Class `Complex`

Overview

Class Properties Class `Complex` has the following properties:

imaginary : EDouble

real : EDouble

4.22.2.3 Class `ContinuousPDF`

Overview

Parent Classes

- ProbabilityDensityFunction (see section 4.22.2.9 on page 179)

4.22.2.4 Class `ContinuousSample`

Overview

Class Properties Class `ContinuousSample` has the following properties:

probability : EDouble [0..1]

value : EDouble [0..1]

4.22.2.5 Class `ExponentialDistribution`

Overview

Class Properties Class `ExponentialDistribution` has the following properties:

rate : EDouble

Parent Classes

- ContinuousPDF (see section 4.22.2.3 on page 178)

4.22.2.6 Class GammaDistribution

Overview

Class Properties Class GammaDistribution has the following properties:

alpha : EDouble

beta : EDouble

Parent Classes

- ContinuousPDF (see section 4.22.2.3 on page 178)

4.22.2.7 Class LognormalDistribution

Overview

Class Properties Class LognormalDistribution has the following properties:

mu : EDouble

sigma : EDouble

Parent Classes

- ContinuousPDF (see section 4.22.2.3 on page 178)

4.22.2.8 Class NormalDistribution

Overview

Class Properties Class NormalDistribution has the following properties:

mu : EDouble

sigma : EDouble

Parent Classes

- ContinuousPDF (see section 4.22.2.3 on page 178)

4.22.2.9 Class ProbabilityDensityFunction

Overview

Parent Classes

- ProbabilityFunction (see section 4.22.2.10 on page 180)

4.22.2.10 Class `ProbabilityFunction`

Overview

4.22.2.11 Class `ProbabilityMassFunction`

Overview

Class Properties Class `ProbabilityMassFunction` has the following properties:

orderedDomain : EBoolean

samples : Sample [0..∗]

Parent Classes

- ProbabilityFunction (see section 4.22.2.10 on page 180)

4.22.2.12 Class `Sample`

Overview

Class Properties Class `Sample` has the following properties:

probability : EDouble [0..1]

value : T

4.22.2.13 Class `SamplePDF`

Overview

Class Properties Class `SamplePDF` has the following properties:

distance : EDouble [0..1]

values : Complex [0..∗]

Parent Classes

- ProbabilityDensityFunction (see section 4.22.2.9 on page 179)

4.23 Package `units`

4.23.1 Package Overview

4.23.2 Detailed Class Documentation

4.23.2.1 Class `BaseUnit`

Overview

Class Properties Class `BaseUnit` has the following properties:

name : String

4.23.2.2 Class `Unit`

Overview

4.23.2.3 Class `UnitCarryingElement`

Overview

Class Properties Class `UnitCarryingElement` has the following properties:

unit : Unit [0..1]

unitSpecification : String

4.23.2.4 Class `UnitLiteral`

Overview

Class Properties Class `UnitLiteral` has the following properties:

baseUnit : BaseUnit

Parent Classes

- Unit (see section 4.23.2.2 on page 181)

4.23.2.5 Class `UnitMultiplication`

Overview

Class Properties Class `UnitMultiplication` has the following properties:

units : Unit [1..$*$]

Parent Classes

- Unit (see section 4.23.2.2 on page 181)

4.23.2.6 Class `UnitPower`

Overview

Class Properties Class `UnitPower` has the following properties:

exponent : Integer

unit : Unit

Parent Classes

- Unit (see section 4.23.2.2 on page 181)

4.23.2.7 Class `UnitRepository`

Overview

Class Properties Class `UnitRepository` has the following properties:

units : BaseUnit [0..$*$]

Chapter 5

Discussion

5.1 PCM versus UML2

Despite the fact that several concepts in the PCM have counterparts in the UML2 meta-model, the PCM's design is intentionally not based on the UML2 meta-model. We argue against commonly used arguments for using UML2 and highlight its additional problems which hinder an approach as described in this paper.

Common arguments for using UML2 as foundation of performance prediction models are the wide-spread use and familiarity of the developers with UML2, the availability of model-instances and the reuse of existing concepts.

We agree that using a notation familiar to developers is necessary to increase the willingness to accept and use a new technique. As a consequence we reused the UML2 graphical notation whenever it appeared adequate. However, as we use model-driven techniques to transform the model instances into prediction models, the source model should be unambiguous in a way which ideally only has one concept to model a certain fact. In UML2 many constructs exist which allow modelling a single fact in many different ways. Take for example a loop modelled in an UML2 activity diagram. Either an iterator node or a control flow going backwards may be used to express it. Facing such ambiguities, transformations turn out to be overly complex as they have to identify all these different options.

The same argument also holds for the reuse of existing UML2 models for performance predictions. Industrial style UML2 models have mostly been designed for human communication. Hence, they use models which need further explanations, UML2 notes, or additional documentation which renders them unsuited for automated, machine interpreted model transformations. Besides the effort of addition performance annotations using a UML profile, such models would need significant effort to prepare them for automated predictions aligning the model with the concepts as expected by the transformation (for further discussions on using UML2 in model-driven approaches see [51]).

Using UML2 tools and profiles for performance annotations raises an additional issue. Performance annotations like those defined in the UML-SPT profile are rather complex and their attachment to model elements is error-prone. Support for this task by means of modern UIs, like on the fly error correction, syntax highlighting, etc. is crucial. However, the UML stereotype mechanism allows only for editing basic datatypes like strings or numbers with very basic editing capabilities. As a way out, some UML tools offer extension mechanisms to customise the tool's editing capabilities. However, such extensions need special coding for every UML2 tool available. In addition to the issues with specifying the annotations, additional problems arise when implementing model transformations using standard transformation approaches like QVT. In present state, support for stereotypes is limited as they do not offer any means to parse tagged values. For SPT based annotations this means transforming the annotations using for example ad-hoc Java transformations which means loosing the advantage of the standardised transformation engine again.

Reusing UML2 concepts sounds good on first sight as well. In our PCM, several concepts like interface, signature, etc. have their counterparts in UML2. However, as UML2 is a large and complex meta-model respecting all kinds of concepts available is difficult - especially, when defining the concept's meaning wrt. a performance prediction model. Therefore, the PCM is restricted to concepts for which we know how to map them onto the performance domain and how to predict their performance impact.

Finally, UML2 is not designed specifically for the aim of doing performance predictions which can be seen by the need of profiles for model annotations. Opposed to that, the PCM includes advanced concepts coming from the CBSE as well as from the performance domain. Examples of the additional concepts are service effect specifications, a component type hierarchy, inherent support for performance annotations and an explicit component context model for expressing a components QoS in dependence of its environment [52]. In this paper, we focus on QoS-relevant modelling elements, a specification of the other concepts can be found on the PCM's website [53].

With all that said, it becomes clear that using the PCM for legacy projects includes migrating existing UML2 model into the PCM. If the models have been designed for human communication, this might not be a problem as the models need checking anyhow. For models which have already been designed for model driven approaches, writing a transformation to initially transform the UML2 model into the PCM becomes necessary.

5.2 Related Work

Component Models In recent years, many component models have been developed for many different purposes. A taxonomy of these models can be found in [7].

- **ROBOCOP:** [54] allows performance predictions, aims at embedded systems.
- **PACC:** [55] allows performance predictions, aims at embedded systems.
- **Koala:** [56] no QoS, aims at embedded systems
- **SOFA:** [21] allows protocol checking.
- **Fractal:** allows runtime reconfigurations of architectures
- **UML:** [57] limited component concept
- **CCM:** no QoS
- **EJB:** no QoS
- **COM+:** no QoS

Performance Meta-Models To describe the performance properties of software systems, several meta-models have been introduced (Survey by [58]).

- **SPE-Metamodel:** [59] designed for object-oriented software systems.
- **UML+SPT profile:** [29] offers capabilities to extend UML models with performance properties. Is not suited for parametric dependencies, which are needed for component specifications.
- **CSM:** [30] is closely aligned with the UML-SPT approach and does not target component-based systems.
- **KLAPER:** [32] is designed for component-based architectures and reduces modelling complexity by treating components and resources in a unified way.
- **UML MARTE:** [37] UML profile with performance annotations for real-time and embedded systems. Includes a hardware model and allocation notations.

5.3 Open Issues and Limitations

Parts of this section are taken from [23].

- **Resource Model:** The PCM's resource model is still very limited and supports only a few types of abstract resource types on the hardware layer. QoS influencing factors from the middleware, the virtual machine, and operating system are still missing in the PCM's resource model. We plan to introduce a layered resource model, where different system levels (such as middleware, operating system, hardware) can be modelled independently and then composed vertically, just as software components are composed horizontally. We also plan to enhance the PCM modelling and analysis capabilities w.r.t. virtualized environments.
- **Dynamic Architecture:** The PCM supports analysing static, component-based architectures with fixed connectors and fixed component allocations to resources. However, some systems (e.g., involving local mobility or web services) have a dynamic architecture, where component connectors can change, components can be allocated to different resources, and components can be replicated at runtime. Methods from Grassi et al. [60] and Caporuscio et al. [61] allow performance prediction and analysis for dynamically reconfigurable architectures. Extending the PCM into this direction would increase the number of analysable systems.
- **Manual Prediction Result Interpretation and Feedback:** The PCM's feedback after running model solvers that deliver performance metrics is limited. Although the automated improvement (cf. Section 3.5) can explore the design space, it does not provide support for manual inspection of the performance characteristics of a single architecture candidate. There are various possibilities for improvement: Tools could graphically highlight bottlenecks in a PCM model instance and annotate the predicted passage times for individuals actions into `System` (combined) and `RDSEFF` instances. Analysis tools could also be prepared for important recurring questions about the performance of a system, such as ”what is the maximum throughput?”, ”what is the bottleneck resource?”, etc., for which they would provide specialised visualisations. Performance solvers could also be adapted to allow an automatic multi-variable sensitivity analysis of the performance of a PCM instance.
- **Internal State:** The internal state of a software component can influence its QoS characteristics in the same manner as input parameter [26]. In the PCM, only a static abstraction of internal state is modelled with component parameters (Chapter 3.4.2.3), which cannot change during runtime. This abstraction avoid state space explosion, because a component-based system can have a huge number of user-dependent internal states. However, future extensions to the PCM could experiment with less high abstractions for the internal state and make it user-dependent and changeable in specific scenarios. For example, state machines could be used to model possible internal states and transitions between them. Component parameter could provide default values capturing the initial state of a component. It remains to be validated in which cases such a model is still solvable.
- **Identification of the Relevant QoS Parameters:** To achieve accurate QoS predictions, the parameters influencing the attributes of interest need to be identified. A lot of work has already been done in this context, in UML for example by the definition of the UML SPT profile [62]. However, the existing work needs to be reviewed, to be extended and the identified parameters needed to be specified within our component model. Furthermore, means to analyse and derive the desired performance metrics from the input values have to be found and/or developed.
- **High-level concurrency modelling constructs:** We plan to add special modelling constructs for concurrent control flow into the PCM. This shall relieve the burden from developers to specify

concurrent behaviour with basic constructs, such as forks or semaphores for synchronisation. The concurrency modelling constructs shall be aligned with known concurrency patterns and be configurable via feature diagrams. Model-transformations shall transform the high-level modelling constructs to the performance domain of analytical and simulation models.

Bibliography

[1] Steffen Becker, Heiko Koziolek, and Ralf Reussner, "Model-based Performance Prediction with the Palladio Component Model," in *Proceedings of the 6th International Workshop on Software and Performance (WOSP2007)*. February 5–8 2007, ACM Sigsoft.

[2] Heiko Koziolek and Jens Happe, "A Quality of Service Driven Development Process Model for Component-based Software Systems," in *Component-Based Software Engineering*, Ian Gorton, George T. Heineman, Ivica Crnkovic, Heinz W. Schmidt, Judith A. Stafford, Clemens A. Szyperski, and Kurt C. Wallnau, Eds. July 2006, vol. 4063 of *Lecture Notes in Computer Science*, pp. 336–343, Springer-Verlag, Berlin, Germany.

[3] Heiko Koziolek, Jens Happe, and Steffen Becker, "Parameter Dependent Performance Specification of Software Components," in *Proceedings of the Second International Conference on Quality of Software Architectures (QoSA2006)*. July 2006, vol. 4214 of *Lecture Notes in Computer Science*, pp. 163–179, Springer-Verlag, Berlin, Germany.

[4] Viktoria Firus, Steffen Becker, and Jens Happe, "Parametric Performance Contracts for QML-specified Software Components," in *Formal Foundations of Embedded Software and Component-based Software Architectures (FESCA)*. 2005, vol. 141 of *Electronic Notes in Theoretical Computer Science*, pp. 73–90, ETAPS 2005.

[5] Ralf H. Reussner, Heinz W. Schmidt, and Iman Poernomo, "Reliability Prediction for Component-Based Software Architectures," *Journal of Systems and Software – Special Issue of Software Architecture – Engineering Quality Attributes*, vol. 66, no. 3, pp. 241–252, 2003.

[6] Clemens Szyperski, Dominik Gruntz, and Stephan Murer, *Component Software: Beyond Object-Oriented Programming*, ACM Press and Addison-Wesley, New York, NY, 2 edition, 2002.

[7] K.-K. Lau and Z. Wang, "A Taxonomy of Software Component Models," in *Proceedings of the 31st EUROMICRO Conference*. 2005, pp. 88–95, IEEE Computer Society Press.

[8] Simonetta Balsamo, Antinisca Di Marco, Paola Inverardi, and Marta Simeoni, "Model-Based Performance Prediction in Software Development: A Survey," *IEEE Transactions on Software Engineering*, vol. 30, no. 5, pp. 295–310, May 2004.

[9] Katerina Goseva-Popstojanova and Kishor S. Trivedi, "Architecture-based approach to reliability assessment of software systems," *Performance Evaluation*, vol. 45, no. 2-3, pp. 179–204, 2001.

[10] John Cheesman and John Daniels, *UML Components: A Simple Process for Specifying Component-based Software*, Addison-Wesley, Reading, MA, USA, 2000.

[11] Steffen Becker, Lars Grunske, Raffaela Mirandola, and Sven Overhage, "Performance Prediction of Component-Based Systems: A Survey from an Engineering Perspective," in *Architecting Systems*

with Trustworthy Components, Ralf Reussner, Judith Stafford, and Clemens Szyperski, Eds., vol. 3938 of *Lecture Notes in Computer Science*, pp. 169–192. Springer, 2006.

[12] Ralf H. Reussner, *Parametrisierte Verträge zur Protokolladaption bei Software-Komponenten*, Dissertation, Fakultät für Informatik, Universität Karlsruhe (TH), Germany, July 2001.

[13] Jim Q. Ning, “A component-based software development model,” in *COMPSAC ’96: Proceedings of the 20th Conference on Computer Software and Applications*, Washington, DC, USA, 1996, p. 389, IEEE Computer Society.

[14] A. W. Brown and K. C. Wallnan, “Engineering of component-based systems,” in *ICECCS ’96: Proceedings of the 2nd IEEE International Conference on Engineering of Complex Computer Systems (ICECCS ’96)*, Washington, DC, USA, 1996, p. 414, IEEE Computer Society.

[15] E. Teiniker, G. Schmoelzer, J. Faschingbauer, C. Kreiner, and R. Weiss, “A hybrid component-based system development process,” in *Proceedings of 31st EUROMICRO Conference on Software Engineering and Advanced Applications*, August 2005, pp. 152–159.

[16] Ivica Crnkovic, Michel Chaudron, and Stig Larsson, “Component-based development process and component lifecycle,” in *International Conference on Software Engineering Advances, ICSEA’06*, Tahiti, French Polynesia, October 2006, IEEE.

[17] David Lorge Parnas, “On the criteria to be used in decomposing systems into modules,” *Communications of the ACM*, vol. 15, no. 12, pp. 1053–1058, Dec. 1972.

[18] Bertrand Meyer, *Object-Oriented Software Construction*, Prentice Hall, Englewood Cliffs, NJ, USA, 2 edition, 1997.

[19] A. V. Aho and J. D. Ullman, *Foundations of Computer Science*, Computer Science Press, W.H. Freeman and Co., New York, 1992.

[20] Ralf H. Reussner, “Automatic Component Protocol Adaptation with the CoCoNut Tool Suite,” *Future Generation Computer Systems*, vol. 19, pp. 627–639, July 2003.

[21] Frantisek Plasil and Stanislav Visnovsky, “Behavior Protocols for Software Components,” *IEEE Transactions on Software Engineering*, vol. 28, no. 11, pp. 1056–1076, 2002.

[22] Ralf H. Reussner, Steffen Becker, and Viktoria Firus, “Component Composition with Parametric Contracts,” in *Tagungsband der Net.ObjectDays 2004*, 2004, pp. 155–169.

[23] Heiko Koziolek, *Parameter Dependencies for Reusable Performance Specifications of Software Components*, Ph.D. thesis, University of Oldenburg, 2008.

[24] Franz Eisenführ and Martin Weber, *Rationales Entscheiden*, Springer Verlag, Berlin, u.a., 4. edition, 2003.

[25] Object Management Group, “Common Object Request Broker Architecture: Core Specification,” March 2004, [letztes Abrufdatum 29.04.2006].

[26] Dick Hamlet, Dave Mason, and Denise Woit, *Component-Based Software Development: Case Studies*, vol. 1 of *Series on Component-Based Software Development*, chapter Properties of Software Systems Synthesized from Components, pp. 129–159, World Scientific Publishing Company, March 2004.

[27] Heiko Koziolek and Viktoria Firus, "Parametric Performance Contracts: Non-Markovian Loop Modelling and an Experimental Evaluation," in *Proceedings of FESCA2006*, 2006, Electronical Notes in Computer Science (ENTCS).

[28] Jens Happe and Viktoria Firus, "Using stochastic petri nets to predict quality of service attributes of component-based software architectures," in *Proceedings of the Tenth Workshop on Component Oriented Programming (WCOP2005)*, 2005.

[29] Object Management Group (OMG), "UML Profile for Schedulability, Performance and Time," January 2005.

[30] Murray Woodside, Dorina C. Petriu, Hui Shen, Toqeer Israr, and Jose Merseguer, "Performance by unified model analysis (PUMA)," in *WOSP '05: Proceedings of the 5th International Workshop on Software and Performance*, New York, NY, USA, 2005, pp. 1–12, ACM Press.

[31] Antonia Bertolino and Raffaela Mirandola, "CB-SPE Tool: Putting Component-Based Performance Engineering into Practice," in *Proc. 7th International Symposium on Component-Based Software Engineering (CBSE 2004), Edinburgh, UK*, Ivica Crnkovic, Judith A. Stafford, Heinz W. Schmidt, and Kurt C. Wallnau, Eds. 2004, vol. 3054 of *Lecture Notes in Computer Science*, pp. 233–248, Springer-Verlag, Berlin, Germany.

[32] Vincenzo Grassi, Raffaela Mirandola, and Antonino Sabetta, "From Design to Analysis Models: a Kernel Language for Performance and Reliability Analysis of Component-based Systems," in *WOSP '05: Proceedings of the 5th international workshop on Software and performance*, New York, NY, USA, 2005, pp. 25–36, ACM Press.

[33] Egor Bondarev, Peter de With, Michel Chaudron, and Johan Musken, "Modelling of Input-Parameter Dependency for Performance Predictions of Component-Based Embedded Systems," in *Proceedings of the 31th EUROMICRO Conference (EUROMICRO'05)*, 2005.

[34] Douglas Schmidt, Michael Stal, Hans Rohnert, and Frank Buschmann, *Pattern-Oriented Software Architecture – Volume 2 – Patterns for Concurrent and Networked Objects*, Wiley & Sons, New York, NY, USA, 2000.

[35] E.D. Lazowska, J. Zahorjan, G. S. Graham, and K. C. Sevcik, *Quantitative System Performance - Computer System Analysis Using Queueing Network Models*, Prentice-Hall, 1984.

[36] Holger Hermanns, Ulrich Herzog, and Joost-Pieter Katoen, "Process algebra for performance evaluation," *Theoretical Computer Science*, vol. 274, no. 1-2, pp. 43–87, 2002.

[37] Object Management Group (OMG), "UML Profile for Modeling and Analysis of Real-Time and Embedded systems (MARTE) RFP (realtime/05-02-06)," 2006.

[38] James A. Whittaker and Michael G. Thomason, "A Markov chain model for statistical software testing," *IEEE Transactions on Software Engineering*, vol. 20, no. 10, pp. 812–824, Oct. 1994.

[39] Kalyanmoy Deb, Samir Agrawal, Amrit Pratap, and T Meyarivan, "A fast elitist non-dominated sorting genetic algorithm for multi-objective optimization: NSGA-II," in *Parallel Problem Solving from Nature PPSN VI*. 2000, vol. 1917/2000, pp. 849–858, Springer-Verlag, Berlin, Germany.

[40] Anne Martens, Heiko Koziolek, Steffen Becker, and Ralf H. Reussner, "Automatically improve software models for performance, reliability and cost using genetic algorithms," in *WOSP/SIPEW '10: Proceedings of the first joint WOSP/SIPEW international conference on Performance engineering*, New York, NY, USA, 2010, pp. 105–116, ACM.

[41] Jing Xu, "Rule-based automatic software performance diagnosis and improvement," *Performance Evaluation*, vol. 67, no. 8, pp. 585–611, 2010, Special Issue on Software and Performance.

[42] John D. McGregor, Felix Bachmann, Len Bass, Philip Bianco, and Mark Klein, "Using arche in the classroom: One experience," Tech. Rep. CMU/SEI-2007-TN-001, Software Engineering Institute, Carnegie Mellon University, 2007.

[43] Vittorio Cortellessa and Laurento Frittella, "A framework for automated generation of architectural feedback from software performance analysis," in *Formal Methods and Stochastic Models for Performance Evaluation, Fourth European Performance Engineering Workshop, EPEW 2007, Berlin, Germany, September 27-28, 2007, Proceedings*, Katinka Wolter, Ed. 2007, vol. 4748 of *Lecture Notes in Computer Science*, pp. 171–185, Springer.

[44] Trevor Parsons and John Murphy, "Detecting performance antipatterns in component based enterprise systems," *Journal of Object Technology*, vol. 7, no. 3, pp. 55–90, Mar. 2008.

[45] Amogh Kavimandan and Aniruddha S. Gokhale, "Applying model transformations to optimizing real-time QoS configurations in DRE systems," in *Proc. of QoSA'09*. 2009, pp. 18–35, Springer-Verlag, Berlin, Germany.

[46] Aldeida Aleti, Stefan Björnander, Lars Grunske, and Indika Meedeniya, "Archeopterix: An extendable tool for architecture optimization of AADL models," in *Proc. of ICSE 2009 Workshop on Model-Based Methodologies for Pervasive and Embedded Software (MOMPES)*. 2009, pp. 61–71, IEEE Computer Society.

[47] Gerardo Canfora, Massimiliano Di Penta, Raffaele Esposito, and Maria Luisa Villani, "An approach for QoS-aware service composition based on genetic algorithms," in *Proc. of Genetic and Evolutionary Computation Conference (GECCO)*, Hans-Georg Beyer and Una-May O'Reilly, Eds. 2005, pp. 1069–1075, ACM.

[48] Lars Grunske, Peter A. Lindsay, Egor Bondarev, Yiannis Papadopoulos, and David Parker, "An outline of an architecture-based method for optimizing dependability attributes of software-intensive systems," in *WADS*, Rogério de Lemos, Cristina Gacek, and Alexander B. Romanovsky, Eds. 2006, vol. 4615 of *Lecture Notes in Computer Science*, pp. 188–209, Springer-Verlag, Berlin, Germany.

[49] Daniel A. Menascé, John M. Ewing, Hassan Gomaa, Sam Malex, and João P. Sousa, "A framework for utility-based service oriented design in SASSY," in *Proc. of Proceedings of the first joint WOSP/SIPEW International Conference on Performance Engineering (WOSP/SIPEW)*. 2010, pp. 27–36, ACM.

[50] Christian Blum and Andrea Roli, "Metaheuristics in combinatorial optimization: Overview and conceptual comparison," *ACM Computing Surveys*, vol. 35, no. 3, pp. 268–308, 2003.

[51] Brian Henderson-Sellers, "UML - the good, the bad or the ugly? perspectives from a panel of experts," *Software and System Modeling*, vol. 4, no. 1, pp. 4–13, February 2005.

[52] Steffen Becker, Jens Happe, and Heiko Koziolek, "Putting Components into Context - Supporting QoS-Predictions with an explicit Context Model," in *Proceedings of the Eleventh International Workshop on Component-Oriented Programming (WCOP'06)*, Ralf Reussner, Clemens Szyperski, and Wolfgang Weck, Eds., 2006.

[53] Palladio, "The Palladio Component Model (Download and Documentation)," http://sdqweb.ipd.kit.edu/wiki/Palladio_Component_Model, December 2009.

[54] Egor Bondarev, Johan Muskens, Peter H. N. de With, Michel R. V. Chaudron, and Johan Lukkien, "Predicting real-time properties of component assemblies: A scenario-simulation approach," in *Proc. 30th EUROMICRO-Conference*, 2004, pp. 40–47.

[55] Scott A. Hissam, Gabriel A. Moreno, Judith A. Stafford, and Kurt C. Wallnau, "Packaging Predictable Assembly.," in *Component Deployment, IFIP/ACM Working Conference, CD 2002, Berlin, Germany, June 20-21, 2002, Proceedings*, Judy M. Bishop, Ed. 2002, vol. 2370 of *Lecture Notes in Computer Science*, pp. 108–124, Springer.

[56] Rob van Ommering, Frank van der Linden, Jeff Kramer, and Jeff Magee, "The koala component model for consumer electronics software," *Computer*, vol. 33, no. 3, pp. 78–85, 2000.

[57] Object Management Group (OMG), "Unified Modeling Language Specification: Version 2, Revised Final Adopted Specification (ptc/05-07-04)," 2005.

[58] Vittorio Cortellessa, "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*, New York, NY, USA, 2005, pp. 195–204, ACM Press.

[59] Connie U. Smith and Lloyd G. Williams, *Performance Solutions: A Practical Guide to Creating Responsive, Scalable Software*, Addison-Wesley, 2002.

[60] Vincenzo Grassi, Raffaela Mirandola, and Antonino Sabetta, "A model-driven approach to performability analysis of dynamically reconfigurable component-based systems," in *WOSP '07: Proceedings of the 6th international workshop on Software and performance*, New York, NY, USA, 2007, pp. 103–114, ACM.

[61] Mauro Caporuscio, Antinisca Di Marco, and Paola Inverardi, "Model-based system reconfiguration for dynamic performance management," *Journal of Systems and Software*, vol. 80, no. 4, pp. 455–473, 2007.

[62] Object Management Group (OMG), "UML Profile for Modeling Quality of Service and Fault Tolerance Characteristics and Mechanisms," May 2005.