Model Transformations in Non-functional Analysis

Steffen Becker

Heinz Nixdorf Institute,
Department of Computer Science, University of Paderborn,
D-33102 Paderborn, Germany
steffen.becker@upb.de
http://www.cs.uni-paderborn.de/en/research-group/software-engineering/people/steffen-becker.html

Abstract. The quality assessment of software design models in early development phases can prevent wrong design decisions on the architectural level. As such wrong decisions are usually very cost-intensive to revert in late testing phases, model-driven quality predictions offer early quality estimates to prevent such erroneous decisions. By model-driven quality predictions we refer to analyses which run fully automated based on model-driven methods and tools. In this paper, we give an overview on the process of model-driven quality analyses used today with a special focus on issues that arise in fully automated approaches.

Keywords: Model-driven quality analyses, performance, reliability, MARTE, Palladio Component Model.

1 Motivation

Dealing with non-functional requirements is still a major challenge in today's software development processes. In the industrial state-of-the-art, non-functional requirements are often not collected in a systematic manner or disregarded during the design and implementation phases. However, in testing phases or, even worse, during final operation on the customer's side, systems often fail due to insufficient performance or reliability characteristics.

Only few examples of insufficient quality have been reported in detail as they are potentially hurtful to the image of the reporting companies. From the available case studies, we reference one from the area of performance problems. The migration of SAP R3 to SAP's ByDesign SOA solution almost failed as the legacy system architecture was unable to work properly in the new environment [34]. The consequence was that the performance was unacceptably low and the system went through a costly redesign process deferring product release by approximately 3 years.

During the last decade, model-based and model-driven quality analysis methods have been developed by the scientific community to prevent such issues. These methods aim at early design time estimates of quality properties like

M. Bernardo, V. Cortellessa, and A. Pierantonio (Eds.): SFM 2012, LNCS 7320, pp. 263–289, 2012.
© Springer-Verlag Berlin Heidelberg 2012

performance or reliability. Based on these estimates, system designs which are unable to fulfil their requirements can be ruled out and thus, costly failures at the end of the development process are prevented. For performance, most approaches are based on the initial idea of software performance engineering as introduced by Smith et al. [44]. Recently these efforts have been consolidated in the book by Cortellessa et al. [13]. For reliability, Gokhale [20] presents a survey on the recent trends.

In this article, we are going to present an overview of model-driven quality analysis approaches. These approaches are specialisations of model-based quality analysis methods. In model-driven quality analysis approaches, the idea is that the software model is the first-class entity under development and all other artefacts should be automatically derived from the model. Hence, activities like quality analyses should also be fully automated. In most cases, the automation requirement is realised by model transformations which automatically derive quality analyses models. This high degree of automation poses much higher requirements on the formalisation of the models involved in the process. Additionally, developers of the necessary transformations have to take difficult decisions on necessary abstractions both in the model's structural as well as for stochastic properties. We use the Palladio Component Model throughout this article as a running example of a model-driven quality analysis approach supporting multiple quality properties.

This article is structured as follows. The following section gives an overview of the process implemented in any model-driven quality analysis method discussed in this article. Subsequent sections then highlight specific aspects of this process. Section 3 explains the input models software architects have to create for quality analyses. Section 4 gives a brief overview of commonly used formal analysis models with a focus on performance and reliability predictions. For a model-driven quality analysis, automated transformations derive analysis models from input models as explained in Section 5. Section 6 illustrates how transformations can be used to automatically include performance overheads into analysis models as performance completions. The remaining sections address practical aspects in model-driven quality analyses. Section 7 illustrates how to use model transformations to bridge the differences between different modelling languages. Section 8 briefly surveys the use of reverse engineering techniques to generate input models for model-driven quality analyses. Section 9 gives a short introduction to architecture trade-offs on the basis of different model-driven quality analyses. To demonstrate the usefulness of model-driven quality analyses, Section 10 gives an overview on three different case studies. Finally, Section 11 summarises the topics presented in this article and gives pointers for further reading.

2 Process Overview

This section gives a high-level overview of the process which the software architect follows in the system's design phase to perform model-driven quality analyses. This process is illustrated in Figure 1. In the following we describe

each of the steps in the order of execution followed by the software architect. We assume a classical web- or three tier business information system as system under study in our presentation. However, for other types of systems, the relevance of different aspects of the system model changes, while in general the process remains fixed.

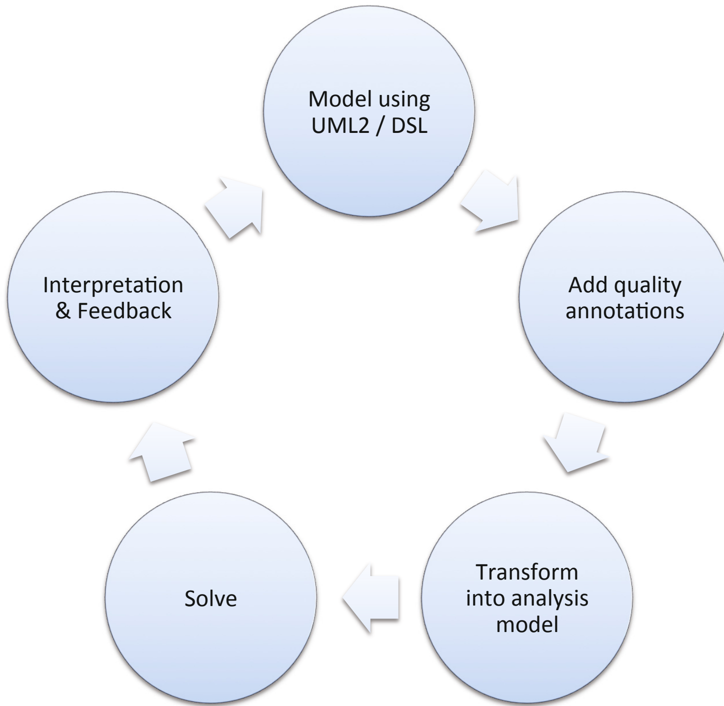

Fig. 1. Process for Model-Driven Quality Analyses

Modelling Using UML2/DSLs: In the first step, software architects needs to create a model of the system under study either using the UML or specific DSLs for software modelling. For most quality attributes this means to create a full model of the system's architecture, i.e., a model that covers a broad range of viewpoints of the system as explained in the following.

Commonly, software architects start by creating a model of the system's static structure, e.g., class or component diagrams. This viewpoint highlights the system's functional blocks and is guided by the functional decomposition of the system's requirements into software elements which realise specific parts of these requirements.

Based on the functional decomposition, software architects specify behavioural aspects of the system under study. Most processes rely on one of two options to represent behaviour: either they specify the behaviour of single entities as

activities of these entities or they rely on the specification of the interaction between different entities by highlighting interaction scenarios, e.g., as sequence diagrams.

Having the system's software parts and their behaviour in place, software architects model the system's deployment. This covers two types of information. First, software architects model the system's hardware environment and its properties, e.g., number and speed of CPU cores, available network bandwidth and latency, disks, memory, etc. Second, architects allocate the entities from the static structure viewpoint to these hardware nodes. In the UML both kinds of information are usually contained in deployment diagrams.

Finally, software architects need to model the use cases of the system. They capture the typical interactions of users with the system. Especially for non-functional analysis use cases need to be significantly enhanced by annotations as explained in the following paragraph.

Add Quality Annotations: The models created in the previous step often contain no or only a limited set of quantitative information which is, however, needed to perform quantitative analyses. For example, for performance analyses, we need to know how much load is generated by each operation of the system on the underlying hardware while serving single requests. Hence, resource demands per hardware type have to be added as quality annotations to the model. Similar, for reliability analyses, we need to know failure rates of each software step and each hardware node.

Also important are probabilities for each use case and the workload they generate on the system. Notice, that the workload specification consists of two elements which we need to annotate: the work and the load. The work is characterised by the complexity of each single job or task submitted to the system, e.g., parameter values or number of elements in collection parameters. The load reflects the frequency of concurrent users arriving at the system and asking for service.

To realise quality annotations, modelling languages follow different approaches. UML2-based approaches rely on profiles. For approximately a decade the SPT profile was used, however, today it has been superseded by the MARTE profile. In case of special DSLs for quality analyses, e.g., the Palladio Component Model (PCM) [7], these languages usually provide build-in mechanisms to specify quality annotations (in the PCM this is the Stochastic Expression language). We provide a more detailed discussion of the quality annotation in Section 3.

Transform: The UML2 or domain specific models are transformed in the next step into analysis models. These models have formal semantics and allow the analysis of properties of interest. For performance, we use queueing networks, stochastic process algebras, or queued Petri-Nets in order to analyse response times, throughput, or utilisations. For reliability analyses, most approaches rely on discrete-time Markov chains. From them, reliability analyses derive for example the probability of failure on demand, i.e., the probability that a certain request issued returns an incorrect result or fails during request processing. For

safety analyses, approaches rely on fault-trees or model checking techniques. For development cost estimates, transformations generate workflow models describing the necessary development tasks and do an estimate of the human resources needed.

To summarise, while the type of analysis model depends on the properties of interest, the overall workflow remains fixed. The software model and its annotations are transformed into formal models which are then solved in the next step.

Solve: After generating the formal analysis models, usually solving these models relies on existing solvers. These solvers are typically either analytical or simulation-based. Analytical methods rely on established mathematical rules which allow fast and accurate analyses of the models they solve. However, this comes at the cost of preconditions the models have to fulfil and limitations on the available result metrics. On the other hand, simulations usually imply no restrictions on the models to be simulated or the result metrics to be collected. However, simulating a model up to a certain level of accuracy takes time. As a rule of thumb, simulations are orders of magnitude slower than analytical methods to achieve the same level of accuracy.

For example, performance models can be solved efficiently with analytic methods if the resource demands or arrival rates are exponentially distributed. For the result metric, they are restricted for example to the mean response time, e.g., the average time a request takes to be handled by the system. If the response time distribution is needed, simulation-based approaches are used.

Interpretation and Feedback: In the last step of the model-driven analysis of a software system, the results collected from solving the analytical models need to be fed back into the initial software model. For example, the utilisation value of a formal representation of a hardware node in the analysis model, e.g., as queue in a queuing network, needs to be interpreted as the utilisation of a hardware node in the initial software model.

Additionally, after feeding the results of the analytical model back into the software model, a decision support step usually takes place which aims at giving recommendations on how to further improve the system's design (or, in case that the results were not satisfying at all, how to make the system's design feasible).

3 Input Models

In the following, we present an example of an input model. We intentionally present an instance of a DSL [18] for non-functional analyses as we assume that UML models with MARTE annotations have been used and discussed more often. We show the Palladio Component Model here as it addresses several requirements:

- Support for a distributed software development process based on a rigorous definition of software components and their functional and non-functional interfaces

- A specifically tailored language known as Stochastic Expression Language to denote QoS annotations
- Explicit support for Model-Driven Completions [2], i.e., in-place transformations of PCM models that automatically include platform specific performance overheads into PCM models (see Section 6)

3.1 Component-Based Quality Analyses

In component-based software development (and similarly in service-oriented development), the development task is split among several independent software developers which develop components or services. However, in software performance engineering, we assume the availability of models of the internal behaviour of the software including the resource demands caused by it according to the process outlined in Section 2. This is a contradiction to the idea that components are provided as black-boxes, i.e., software architects only have access to the component's binary code and its specification. Hence, in order to enable performance engineering in component-based software development processes, component developers have to provide specifications on the performance relevant behaviour of their components.

In the Palladio Component Model, component developers model components (either basic or composite), their interfaces, and their behaviour. For the latter, they use so called `ResourceDemanding-ServiceEffectSpecifications` (RD-SEFFs) [7]. RD-SEFFs model the externally observable behaviour of single component operations as a kind of activity diagram (cf. Figure 2). These activities contain so called `InternalActions`, `ExternalActions`, acquiring and releasing resources, and control flow constructs.

`InternalActions` model the observable resource requests a component issues while running. This includes for example CPU demands as a consequence of algorithmic computations, I/O demands to hard disks, or network accesses. `ExternalActions` model the call of an operation of a component connected to the required interface of the calling component. Modelling `ExternalCalls`, however, requires special attention. As the parameters passed to a called operation may have significant impact on the execution time of the operation, component developers also need to specify performance characteristics of the parameter values. To give an example, if a component operation executes an algorithm on items of a collection, component developers need to provide a specification of the number of elements in this collection (cf. Figure 2 and Section 3.2).

`Acquire-` and `ReleaseActions` model access to limited resources like semaphores, database connections or locks, etc. Control flow constructs model loops, branches, and fork-join blocks.

Most elements may be annotated using the Stochastic Expressions languages as explained in the following subsection.

3.2 Stochastic Expression Language

Stochastic Expressions [7] are used in the PCM as annotation language. Software architects use this language to provide quantitative stochastic information in

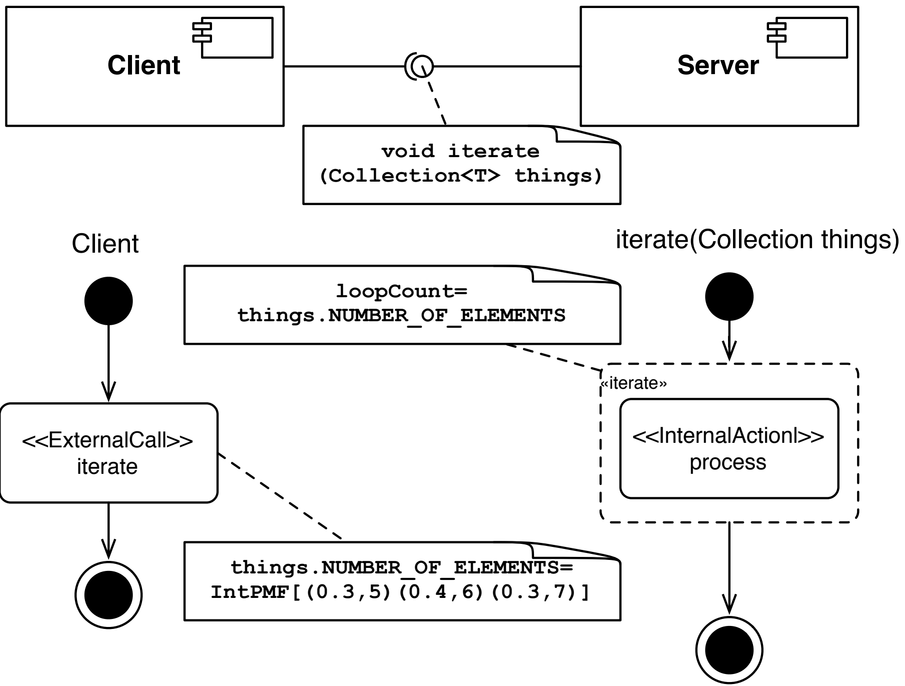

Fig. 2. Example of one component's SEFF calling another component's SEFF

order to characterise typical executions of the software systems they model. For example, software architects can annotate the number of loop iterations, resource demands caused by `InternalActions`, branch transition probabilities, or aspects of the data flow, e.g., how the size of collections changes during processing. An alternative language for a similar task is the value specification language (VSL) from the MARTE specification [39].

In the following, we discuss two examples of stochastic expressions. In the first example, we model the fact that a loop iterates in 30% of all cases 5 times, in 40% of all cases 6 times, and in the remaining 30% it loops 7 times. The corresponding stochastic expression is

$$\texttt{IntPMF}[(0.3, 5)(0.4, 6)(0.3, 7)] \tag{1}$$

`IntPMF` here indicates that the expression is of type Integer and PMF is an abbreviation for probability mass function, i.e., the expression characterises the distribution of a discrete random variable [10].

In the second example, we illustrate the use of stochastic expressions to model performance relevant aspects of the data flow as introduced by Koziolek in his PhD thesis [28]. Assume we model the performance of a component's method which iterates over a collection of items. The signature of the method is

$$\texttt{void iterate}(\texttt{Collection} < \texttt{T} > \texttt{things}) \tag{2}$$

As in component-based software development we do not know the end-user of a component while we are developing it, we cannot make assumptions on the number of elements in this collection. Hence, the caller of the component's method needs to specify this number in the PCM instance. For this, she adds an annotation to the `ExternalCall` that models the call in our example (cf. Figure 2, left-hand side). If we want to encapsulate the same loop we modelled in Example (1) in a component, the annotation on the caller side would be

$$\texttt{things.NUMBER_OF_ELEMENTS} = \texttt{IntPMF}[(0.3, 5)(0.4, 6)(0.3, 7)] \qquad (3)$$

The developer of the component which contains the `iterate` method can use this expression, e.g., to specify the loop count (cf. Figure 2, right-hand side) simply as

$$\texttt{things.NUMBER_OF_ELEMENTS} \qquad (4)$$

This mechanism allows to reuse components and their specification in multiple *contexts*, i.e., in multiple environments which are determined by the component instance's usage, its connected external component instances, and its allocation. The example above illustrates a flexible usage context. However, the PCM also supports flexible allocation contexts as well as assembly contexts using a similar approach [5].

Notice, both examples show the concrete textual syntax of the stochastic expression, i.e., the format in which software architects specify these annotations. To foster the use of model-driven technologies, especially the use of standardised model transformation languages like QVT [38], textual annotations like the stochastic expressions presented here, need to be parsed into an abstract syntax tree and not just stored as simple string values. In the PCM, this is done by a parser generated by the ANTLR framework [40]. This parser has been integrated with the PCM manually. However, today, model-driven approaches like the xText framework [17] are used more frequently to generate concrete textual syntaxes [21].

3.3 Completions

In this subsection, we introduce model-driven performance completions [23]. Performance completions have been introduced by Woodside et al. [46] as a mean to include the performance impact of underlying system layers into performance analyses, i.e, layers below the application layer itself. Examples of sources for such kinds of impact are internal database overheads, overheads of middleware platforms like resources needed to serialise messages, virtual machine layers, etc.

Model-driven performance completions extend the initial idea in situations where the application code is strictly corresponding to its model, e.g., when it is generated from the model. In such cases, model-driven completions are automatically added to the model in order to increase its accuracy. Most approaches use in-place model-transformations to enrich the analysis model automatically.

In this section, we focus on the input models needed for model-driven performance completions, while Section 6 explains the details of the technical realisation of these completions.

In the modelling phase, software architects need to specify their selection of layers they use for their implementation. As an example, consider Figure 3.

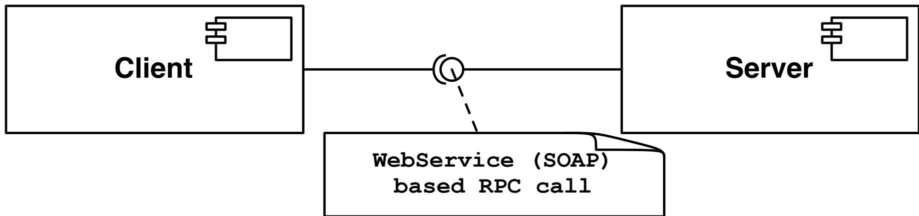

Fig. 3. Annotated component communication

In our example, we see again the `Client` and `Server` component used before. They are connected via an assembly connector. In the technical realisation of this architecture, the software architect has to decide how to realise the communication running over the assembly connector technically. For example, in Java she could select among an RPC realisation based on Java RMI, REST, or the SOAP protocol used in the WebService technology stack. Further options arise if the software architect intended a realisation based on message exchange, e.g., using the Java Messaging Standard (JMS).

If we now consider the performance overhead implied by each variant, we realise that this overhead differs significantly. For example, when we compare RMI and SOAP, SOAP has a much higher overhead due to the fact, that SOAP relies on XML technology. Hence, there is additional time required to process XML documents, send them over the network (as they usually need more bytes to encode their information), and to de-serialise and interpret them on the receiver's side.

For the input models of the quality analysis process, we assume that the software architect is well aware of such technical alternatives. However, as she is not an expert for performance or reliability modelling, she does not want to provide detailed specifications of the quality impact of her choice. Therefore, our model-driven performance completions require the software architect to annotate model elements of the application model (e.g., assembly connectors) with the technical details on their realisation in an abstract way.

In our approach, we selected feature diagrams [14] and their instances (known as configurations) as a well-known modelling formalism to express different variants in software product line engineering. To give an example, Figure 4 shows a feature diagram for the selection of the communication details.

The feature diagram first allows to differentiate between local and remote procedure calls (with local calls having almost no performance impact). For remote procedure calls, software architects can select among RMI and SOAP. Additionally, they can add features which impact the reliability, performance or security of the connector.

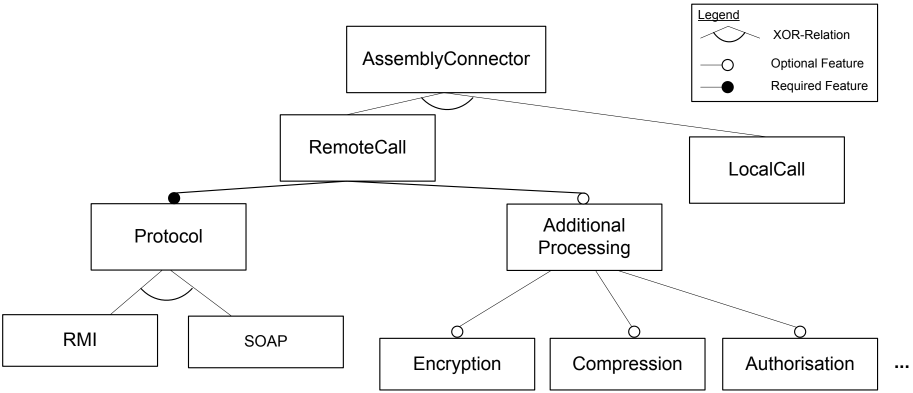

Fig. 4. Feature diagram showing different options to realise an assembly connector [3]

In order to create a valid input model for the model-driven quality analysis, a software architect needs to select a feature configuration for all connectors in the model.

4 Analyses Models

In the following we briefly introduce analysis models for performance and reliability. They are the output of the transformations discusses in the next Section. We can distinguish analysis models according to the quality properties they are able to analyse.

For **performance**, a large set of analysis models has been developed in the past - each of them having its own advantages and disadvantages. In general we aim for models which can be analytically solved which normally implies efficient solutions that can be solved quickly. However, if models become too complex, we have to simulate which is time intensive.

Among the often used models in analytical performance predictions are queuing networks [10]. They are intuitive to understand, and many networks can be solved analytically with a sufficient degree of accuracy with respect to mean response times, utilisation and throughput. However, their disadvantage is while they provide a good abstraction of shared resources like CPUs, HDDs, or network links, they fail in modelling the layered behaviour of software systems, i.e., blocking calls in client/server communications, or acquiring and releasing passive resources. On the tooling side, there is a variety of tools, for example, the Java Modelling Tools which both provide analytical and simulation based solvers.

To overcome these limits, Layered Queuing Networks have been introduced [42]. They allow to model systems in layers, where the behaviour of an upper layer is allowed to call the behaviour of lower layers. The lowest layers then contain the hardware resources as in standard queuing networks. LQNs have built-in support for performance relevant aspects of client/server systems like thread

pools or clean up procedures on the server side that execute after sending the request's response. Heuristic and simulation-based solvers exist that can predict mean response times, utilisations and throughput of systems efficiently.

For concurrent systems, Queuing Petri nets extend standard stochastic Petri nets [1]. They introduce a new type of places that model queues, i.e., tokens get queued when they enter such places and can only leave the place after passing the queue. They provide an easily understandable modelling language. However, most Queuing Petri nets cannot be solved analytically but need to be simulated to derive response time distributions, utilisations of queuing places, and net throughput. Recent tools like the QPME tool rely on Eclipse technologies for modelling and analyses.

Finally, there are stochastic process algebras which model systems as a set of communicating processes. Among them, for example, PEPA [24] is a process algebra which has been applied in different projects successfully. The advantage of this formalism is that many systems are indeed implemented as a set of communicating systems. However, process algebras are often difficult to use for systems where you have to model a large set of identical processes, e.g., the user processes in business information systems. PEPA is supported by tools implemented on top of Eclipse technologies.

For **reliability** analyses, most approaches are built on top of Discrete Time Markov Chains (DTMCs) [10]. In DTMCs, time passes in discrete steps. The Markov chain itself is composed from states and probabilistic transitions. On each time step, the Markov chain proceeds to the next state according to the transition probabilities of all outgoing transitions of the current state. DTMCs can be solved efficiently by analytical means. Today, most approaches utilise the PRISM probabilistic model checker [25] to solve the DTMCs. PRISM reads the DTMC to be analysed in a textual language defined by the tool. It then can evaluate for example the steady state probability to reach a failure state from the DTMC's start state.

When taking the viewpoint of model-driven performance analyses, we notice, that most languages and tools discussed above have been developed before current model-driven methods and tools. Hence, most of them do not provide a MOF [37] compliant meta-model which is a pre-requisite for the use of most model-driven tools. The reason is that these tools have to rely on a common modelling foundation and data formats. As a consequence, we cannot simply transform our software models to analytical models via model-2-model transformations, but need an additional model-2-text transformation step that generates the corresponding performance model in a format readable by the solvers. However, such transformation and parsing steps incur additional performance overheads for solving the performance models. They are crucial at least in online methods [13], i.e., when we use performance predictions at run-time (cf. Section 11).

Second, we notice that transformations from software models to analytical model may need to make additional abstraction steps in order to keep the analytical models solvable in reasonable time. This may both include abstractions in the structure of the model as well as in the complexity of the annotations.

To give an example, in the PCM there is an additional transformation step involved (known as Dependency Solver [11,28]) that computes for each component instance its context and the impact of this context on the component's performance or reliability. This involves computing the convolution of probability distributions which is realised via Fourier transforms. They are computational complex. The same problem arises when having general MARTE annotations in the software model and generating analytical model with tighter constraints. For example, MARTE allows the use of generally distributed service time specifications while most analytical performance models are restricted to exponential distributions to ensure efficient analytical solutions.

5 Transformations

In this section, we give concrete examples for transformations for the performance and the reliability domain. We discuss on these examples which requirements they have to deal with. We use the transformations implemented in the Palladio Component Model for the discussion here as typical examples of transformations used in other approaches. For an overview, see Figure 5.

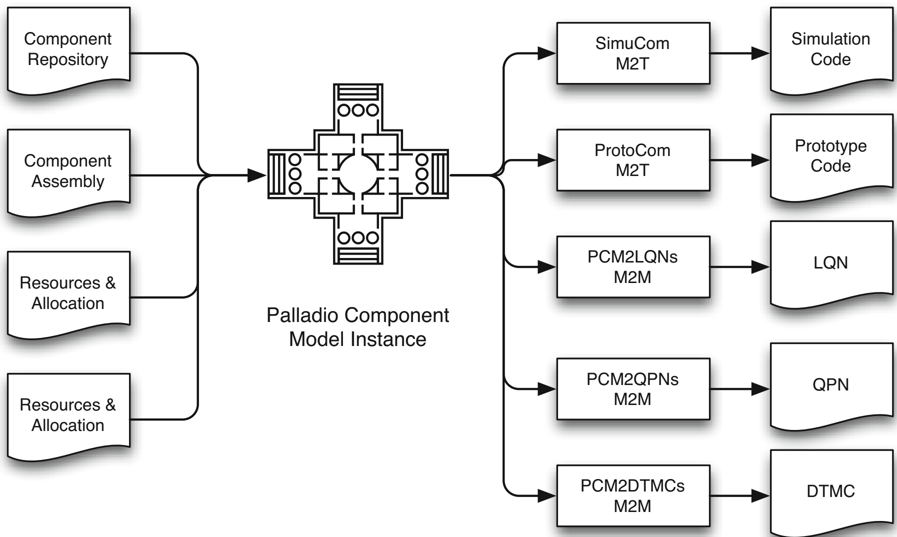

Fig. 5. Overview of PCM transformations

5.1 PCM2SimuCom

SimuCom (Simulation of Component Architectures) is the PCM's reference solver. It has been implemented as a simulation to define and evaluate the semantics of PCM models with respect to their performance properties. It relies on an extended

queuing network simulation, where the PCM's active resources (CPUs, HDDs, LANs) are mapped to simulated queues. The behaviour as defined by the RD-SEFFs of the components is simulated directly. SimuCom also directly interprets Stochastic Expressions by drawing samples from the simulation framework's random number generator. Figure 6 gives an overview on the layers of a SimuCom simulation.

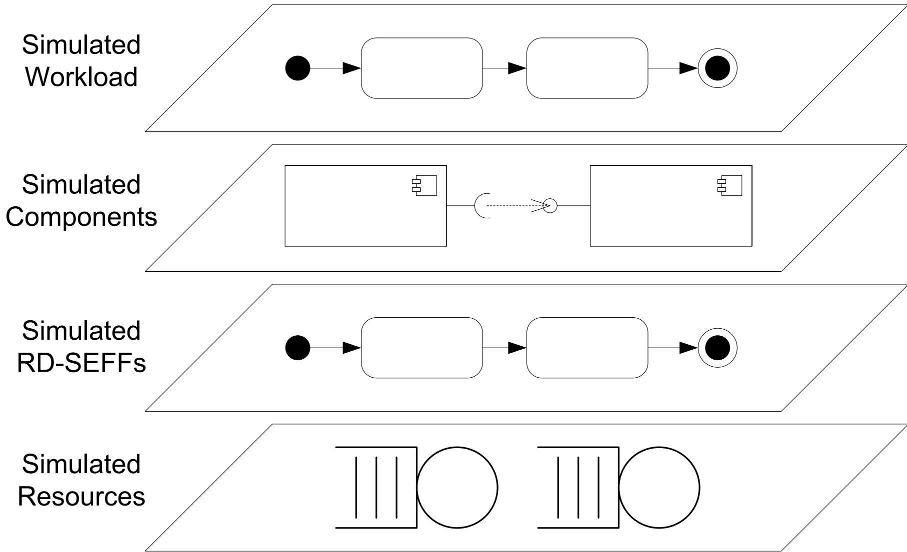

Fig. 6. Conceptual overview on SimuCom [3]

On the top layer, SimuCom simulates users accessing a component-based system. For each user, it simulates the user's behaviour. Each call to the simulated system then triggers a control flow thread that runs through the components and triggers their RD-SEFFs. Finally, `InternalAction`s inside of RD-SEFFs cause resource loads which are then handled by the simulated resource queues.

In the context of this paper, we focus on the model-driven realisation of SimuCom. SimuCom simulations are generated by a model-driven tool chain as Java-based simulation code (cf. Figure 7). This tool chain is an instance of the Architecture Centric Model-Driven Software Development (AC-MDSD) paradigm introduced by Völter and Stahl [45]. In this paradigm, software is generated in a single model-2-text transformation step. This steps reads an instance of a DSL (here a PCM instance) and generates source code of it which makes use of generic library code (also known as platform code in AC-MDSD).

In SimuCom's transformation, we generate components as specified in the PCM instance. For a component, we use a set of implementing Java classes. The methods of these classes contain the code to simulate the component's RD-SEFF. SimuCom's platform contains generic simulation support code. It encapsulates

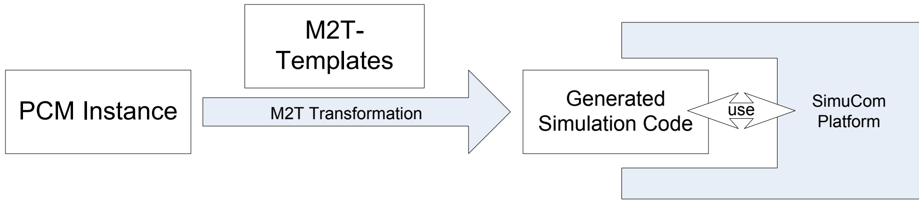

Fig. 7. Overview on SimuCom's transformation [3]

the supported simulation frameworks (Desmo-J [16] and SSJ [32]) and provides generic high-level functions. The latter include evaluating stochastic expressions, the simulation logic of queues, and the code to instrument the simulation in order to collect the metrics of interest. In the PCM's tool, generated simulations are compiled and executed on-the-fly.

5.2 ProtoCom

Using the same underlying principles as SimuCom, ProtoCom [4] (Prototyping of Component Architectures) generates Java source code from PCM instances that can be used as performance prototypes. Here, the idea is to create artificial load which represents the load specified in the PCM instance on real hardware. This has the advantage that a realistic infrastructure environment is present and no restrictions due to model abstractions apply. Examples of such abstraction used in today's performance analyses models are: disregard of memory limitations and memory bandwidth, abstraction from details of the underlying middleware, realistic operating system scheduling policies, hard-drive characteristics, etc. However, the major disadvantage of performance prototypes is that it takes a lot of time to configure and run them on realistic soft- and hardware environments.

Generating prototypes using model-driven techniques relieves developers from the burden to develop such prototypes by themselves. ProtoCom follows the same AC-MDSD process as SimuCom, i.e., again there is a single model-2-text transformation that generates the prototype based on a given PCM instance. However, the platform code now contains algorithms which can be used to mimic resource loads on CPUs and HDDs instead of the simulation code.

5.3 PCM2LQNs

PCM2LQN [29] allows the generation of LQN models from PCM instances. The layers provided by the LQNs are used in this transformation to reflect the layers implied by PCM models (cf. Figure 6). That is, there is a layer for user behaviours, for the system, and for each of the component instances. Like in the queuing network based SimuCom, hardware resources are again mapped to queues. For full details of the PCM2LQN mappings please consult [29].

From the model-driven perspective, PCM2LQN is a model-2-model transformation. It is implemented as a two step transformation. In the first step, the

dependencies are resolved as explained in the beginning of Section 4. In the second step, the LQN model is created. Both transformations are implemented in Java and use the Visitor design pattern [19] to structurally [15] traverse the PCM instance and generate elements in the LQN.

5.4 PCM2QPNs

As a last example from the performance domain, we introduce PCM2QPN [36]. This transformation takes a PCM model and generates a Queuing Petri net. The central mapping ideas are as follows. User requests are represented as tokens in the Petri net. These tokens traverse through the network of places and transitions. Places represent single elements of PCM RD-SEFFs. In case of control flow elements of RD-SEFFs, transitions are aligned according to their RD-SEFF counterparts. For example, for a loop there is a place which models checking the loop condition and a network of places and transitions which models the loop's body behaviour.

Internal actions are represented by queued places in the QPN. Their scheduling discipline is set according to the scheduling discipline of the active resource referenced by the internal action.

Investigating the transition again from a model-driven perspective, it is similar to the PCM2LQN transformation. Again, there are two steps, where the first transformation step solves the dependencies of single component's quality annotations. The second step then generates the QPNs using the solved dependencies. It is implemented as a model-2-model transformation using the imperative QVT Operations [38] transformation language.

5.5 PCM2DTMCs

In the area of reliability predictions, we present the PCM2DTMC approach [11]. Conceptually, it generates DTMCs from a given PCM instance. In these DTMCs it takes both, hardware and software failures into account. For the hardware failures, a matrix is computed which contains the probabilities for all possible combinations of any hardware device from the PCM model being either in a working or in a failure state. For example, if we consider two hardware resources, e.g., two CPUs, we have four possible hardware states. Either both CPUs are working, both failed, CPU1 failed but CPU2 is working, or CPU1 is working but CPU2 failed.

For software failures, RD-SEFFs are interpreted as control flow graphs, where each action in the RD-SEFF can either succeed or fail - the latter either due to a software failure or due to a hardware failure. In case of hardware failures, only failures of hardware resources needed for a certain step in the RD-SEFF are taken into account. For all resources needed in a step, the transformation sums up the probabilities of all system states in which any needed resource is in a failure state.

Inside the DTMC, the control flow graph given by the RD-SEFFs is translated into a Markov chain where each state in the Markov chain models the fact,

that the corresponding RD-SEFF is processing the corresponding action. Then for each of these states in the Markov chain, there is an additional transition to a single failure state. This transition has the combined probability of a software or hardware failure which would lead to a failure in the processing of this action. Figure 8 gives a simplified example of a RD-SEFF with failure probability annotations for software failures on the left hand side. The right hand side illustrates the resulting DTMC showing the two absorbing states for failure and success and the two transient states which represent the two RD-SEFF's `InternalActions`.

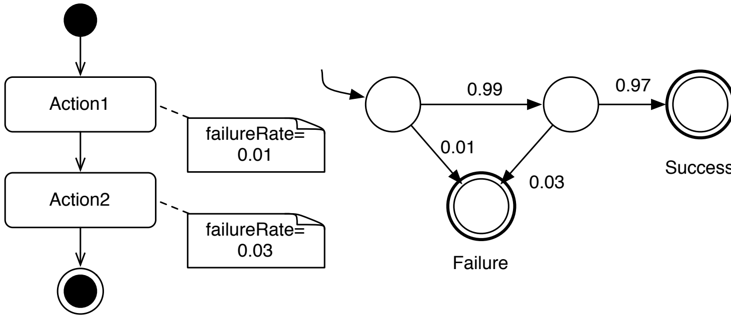

Fig. 8. Illustrating example of an RD-SEFF and its corresponding DTMC

PCM2DTMC generates a DTMC for each possible hardware state from the matrix of all hardware states. For each DTMC it computes the probability of failure on demand (PROFOD).

From the perspective of model transformations, the PCM2DTMC approach uses again a two step transformation approach. In the first step, component dependencies are solved and the hardware failure matrix is computed. In the second step for each hardware system state a DTMC is generated and solved. The transformation is a structural transformation written in Java.

6 Model-Driven Completions

As introduced in Section 3.3, model-driven completions enhance performance models with details which model performance overheads of underlying infrastructure services. In the following, we focus on connector completions and we restrict the discussion to performance as quality attribute.

Types of Completion Transformations: In principle, there is a general choice in designing model-driven completions. First, we can use a transformation to alter the software model in-place and then use standard transformations to create the analysis models from the extended software model. Second, we can enhance the transformation from the software model to the analysis model and create an extended analysis model. The advantage of the first alternative is that

we can reuse all features of the software modelling language, e.g., the introduced stochastic expressions from the Palladio Component Model. Furthermore, the completion transformation often does not become too complex, as it can be split into a part that adds completions and a reused part that generates the analysis model. The advantage of the second alternative is that we have direct access to all features provided by the analysis model, i.e., we can utilise special modelling features like modelling resource demands which happen after sending the response in LQNs (cf. Section 4).

In the PCM, we use the first alternative as its advantages outweigh its disadvantages in our context. Especially the fact that PCM supports multiple analysis transformations which we all could reuse in the first approach is a strong argument.

Connector Completions in the PCM: In the following, we illustrate how the PCM's completion transformations include technical details of RPC connectors as an example. As input we expect PCM models which have been annotated by the software architect as illustrated in Figure 3 in Section 3.3. The aim is to include PCM components that model the performance overhead of the middleware platform which realises the annotated connectors.

To design the completion, we first need to understand the reasons for the performance overhead of RPC communication. First, the method call and all its parameters are sent to the server. For this, they are processed by a pipe-and-filter architecture [12], that takes the high level method call and generates its serialised form as byte stream. It first marshals the method call and all parameters which causes respective computational performance overheads. Depending on additional setting for the connector, subsequent filters in the pipe-and-filter chain encrypt the message, sign it, validate it, etc. Depending on the message size, this causes again a performance overhead. When the communication layer has produced the byte stream, it is send over the network which delays the processing according to network throughput and latency. Notice, that the size of the network package depends on the underlying RPC protocol and all applied processing steps. For example, SOAP messages are usually larger than RMI messages, causing an additional networking overhead. On the server side the whole process is executed in the other direction, i.e., the byte stream is converted back into a method call. Finally, after processing the method call, the whole RCP stack is used again to send back the server's response.

We now present the in-place transformation [15] we have created to model this processing chain as performance completion. In the first step, we remove the annotated connector and replace it with components which model the marshalling and demarshalling steps. The result is shown in Figure 9.

Figure 9 shows the connector completion component (as indicated by the component's stereotype) which now replaces the connector in Figure 3. It has the same provides and requires interface as the connector it replaces (`IA` in our example). This is needed to fit our completion in the place of the original connector and still create a valid PCM instance. The second aspect to notice are the interfaces starting with `IMiddleware`. They have been introduced to

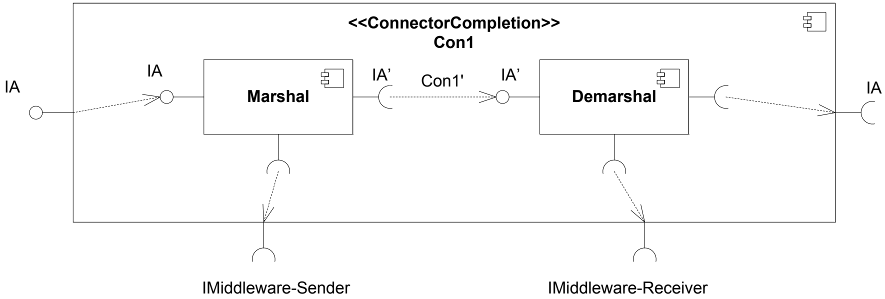

Fig. 9. First transformation step to include a connector completion [3]

make the completion more flexible. The underlying idea is that the `Marshal` and `Demarshal` components do not issue resource demands directly, but rather delegate their internal processing steps to the `IMiddleware` interfaces. These interfaces then are connected to a specific instance of a middleware component, e.g., a Sun Glassfish JavaEE server component. By exchanging this middleware component, we can model the different performance overheads of different server implementations. For example, exchanging the Glassfish component by a JBoss component we can include the performance overhead of this particular JavaEE server into our model. Also notice, that there are a client and a server variant of the middleware interface. Hence, we can also include different overheads on each communication side, depending on the real deployment setup.

In order to include the performance overhead of further processing steps, the transformation now executes further steps depending on the selected features in the connector's annotation. For example, assume the software architect in addition to the communication protocol also selected an encrypted communication channel. Then an additional step, as illustrated in Figure 10, includes the additional overhead into the completion component by adding another completion component, i.e., by applying the transformation idea illustrated in the previous paragraph again.

In Figure 10 we can again see the same structure we discussed before. However, notice that the interface has changed from `IA` to `IA'`. This was needed as the `IA'` interface now deals with the byte stream which represents the method call on the network instead of the method call itself. The interface `IA'` takes track of the size of the message stream as this size is important for the network overhead later (as stated in the description of the RPC protocol above). However, as the full details of the structure of the interface `IA'` would go into too much detail, we refer the reader to [3] where the full mechanics of the transformation are documented.

Figure 11 shows the resulting model after the execution of the completion transformation.

In Figure 11 we can see that the transformation has added a component realising the middleware on both, the client's and the server's, side. Which middleware

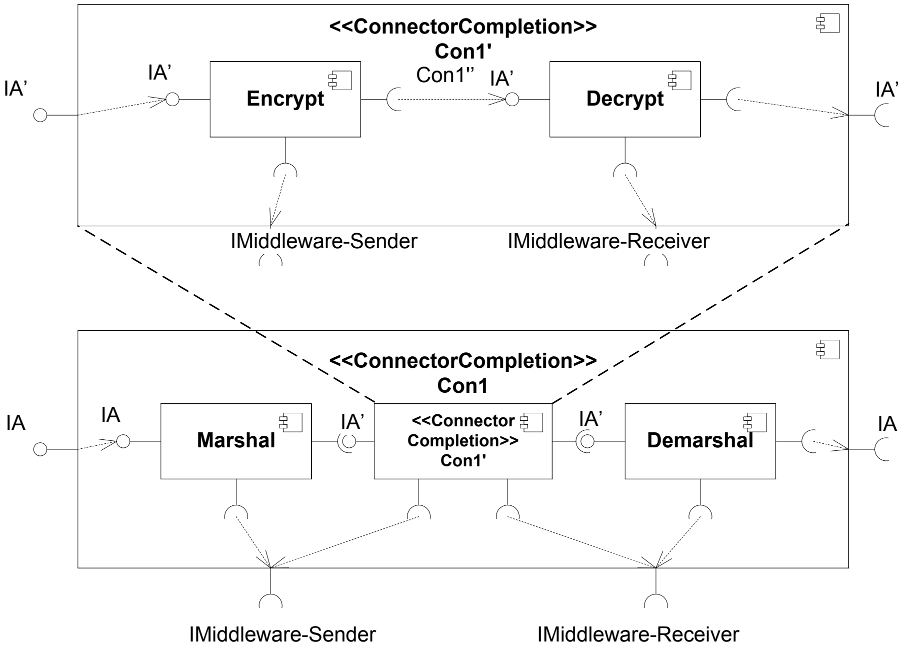

Fig. 10. Second transformation step to include a connector completion [3]

component gets added on which side is configurable in the system's deployment model. However, providers of middleware server's ideally need to provide a library of components which model their server's performance overhead. Today such libraries are not available. As an alternative, we used an automated benchmark for our experiments in [23] which created performance models of middleware servers. Our completion transformation then included these generated models. Additionally, we can see in Figure 11 that the completion transformation automatically deployed the created components to either the client's deployment node or the server's node.

To summarise, we have demonstrated in this section how a completion transformation transforms specially annotated software models to include middleware

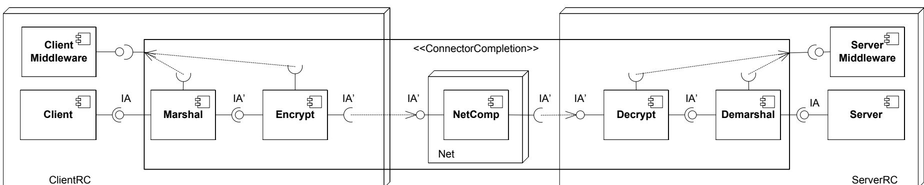

Fig. 11. PCM instance with connector completion [3]

details. Besides this special use case, the general principle on how to design a completion transformation is invariant of both the type of completion as well as the quality attribute under study.

7 Model-Driven Integration

In software quality analyses, model transformations are not only used to transform software design models into analysis models, but also to bridge different levels of abstraction more easily or to integrate different software architecture description models.

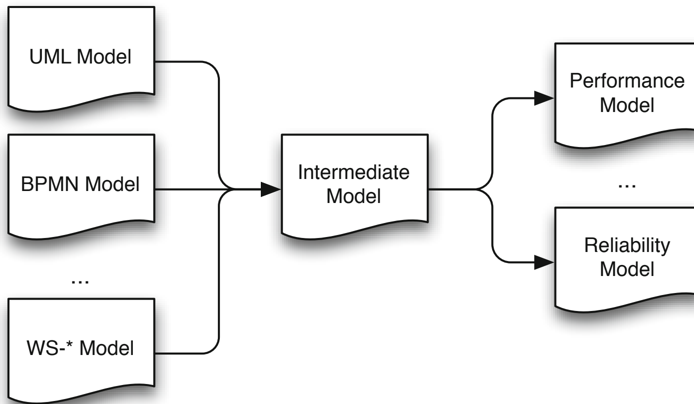

Fig. 12. Basic idea of intermediate models

Intermediate Models: There are approaches like KLAPER [22] or the Core Scenario Model (CSM) [41] which promote the use of intermediate models. These models aim at alleviating the creation of transformations into the analysis domain. To reach this goal, they favour a two-step transformation from the software model to the analysis model. First, a software model is transformed into the intermediate model which then is transformed into the analysis model.

Figure 12 outlines the idea of intermediate models. Their advantage is that they allow to transform n different software models into m different analysis models by just $n + m$ different transformations (in contrast to $m * n$ transformations if each software model is transformed into each analysis model directly). However, the disadvantage is that intermediate models may become quite complex in order to support all features available in all software modelling languages. For example, none of the cited approaches in this article is able to deal with the full complexity of UML, they just allow the use of a subset of it.

Integration of Software Models. There are also some approaches for using model transformations to integrate different types of software modelling

languages. The DUALLy tool developed by Malavolta et al. [33] focuses on bridging different software architecture modelling languages. In the EU project Q-ImPrESS [6] we focused on bridging different component-based software modelling languages via a core model that takes quality aspects into account. Figure 13 provides an overview on the approach taken by Q-ImPrESS.

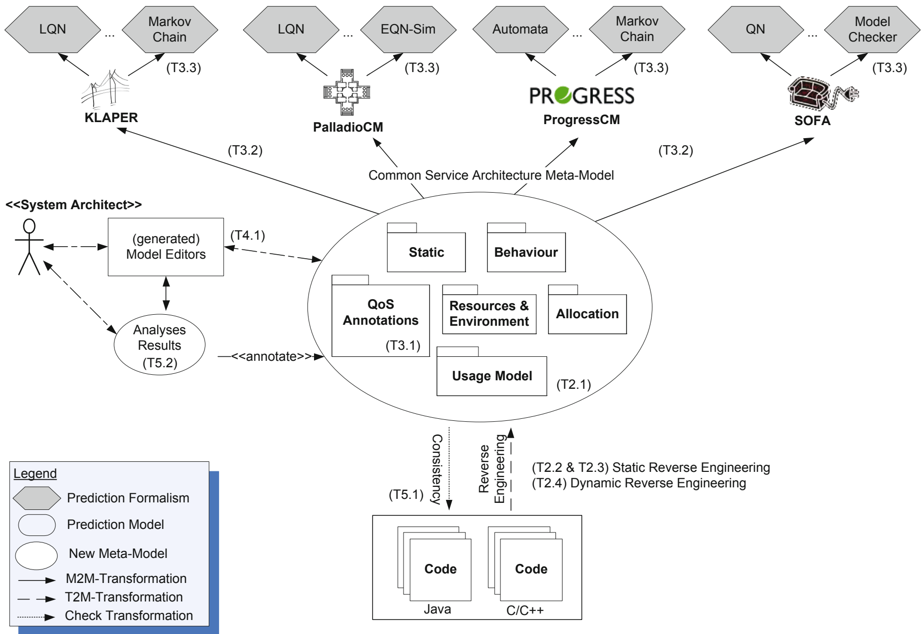

Fig. 13. Overview on Transformations in Q-ImPrESS [8]

There are different software architecture modelling languages on top of Figure 13. Each of these approaches has its own dedicated meta-model and support different types of analyses. To reuse as many of these analysis approaches, Q-ImPrESS promotes the use of a core meta-model, the Service Architecture Meta-Model (SAMM). Transformations exist from the SAMM into each of the software architecture models. This allows to perform different analyses with respect to different types of quality properties. This is useful for architecture trade-off analysis, i.e., finding the right trade-off among different quality attributes (see Section 9) for details.

On the bottom of Figure 13, it is also indicated that Q-ImPrESS also supports reverse engineering of source code to create initial instances of the SAMM. This step is discussed in the following Section.

8 Reverse Engineering of Models

In model-driven quality analyses, the most important step is to get to the models of the software under study. Hence, researches have proposed approaches in the

past to create initial models from source code or execution traces. However, only a few of them support reverse engineering of models which can be used for quality analyses.

For Q-ImPrESS and the PCM, there is support via the SoMoX tool chain [30]. SoMoX creates instances of the SAMM from object-oriented source codes written in Java, C++ or Delphi. It analyses the source code and clusters its classes into components. This is done according to a well-defined set of software metrics specifically tailored to identify components. For components detected in this way SoMoX automatically creates RD-SEFFs (as introduced in Section 3). These RD-SEFFs are limited to `ExternalCall`s and control flow elements.

One step further goes Beagle which builds on top of SoMoX [31]. It executes the software components in typical use cases, e.g., test cases, and counts the byte code instructions the Java VM executes while running the component. Based on these counts, Beagle approximates stochastic expressions using a genetic programming heuristic. This heuristic fits observed byte counts to estimated stochastic expressions and tries to find the best fit.

9 Architecture Trade-Offs

With model-driven quality analyses in place, software architects are able to analyse a particular software system's model for multiple quality attributes like performance, reliability, or costs. Consequently, the question arises how to decide among two different software architectures, i.e., how to trade-off the different quality properties.

Here, two types of approaches exist in literature. First, **manual approaches** where software architects interpret the analyses results and try to identify the underlying trade-offs in discussions among the stakeholders. Especially, SAMM [27] and ATAM promoted by the SEI contain structured processes for this. Second, **quantitative methods** which are semi-automated and supported by tools try to quantify the preferences of the software architect for different quality properties and compute a ranking of alternative software designs according to these preferences. In the latter, the AHP method invented by Saaty has been used [43]. We describe this approach in the following.

In the AHP process, the decision problem is split into hierarchical subdecisions. For example, we could subdivide the overall quality goal into external quality goals (e.g., performance, reliability) on the same level as the goals for internal quality attributes (e.g., maintainability, understandability). For each hierarchy level, the AHP method determines weights that capture the decision makers preferences for all sub-goals. For example, if the decision maker states that internal and external quality goals are equally important, the weights would be 0.5 (50%) for each of the sub-goals. Finally, different software architecture alternatives are evaluated with respect to the leafs of the decision tree and AHP computes from this an overall ranking of the alternatives.

10 Case Studies

In this section, we report on case studies of model-driven quality analyses in practice. All case studies are based on Q-ImPrESS or PCM. The reason for this may be in the fact that Q-ImPrESS and PCM have mature tooling which allows their use in industry projects. Most other model-driven quality analysis approaches found in literature stay, in contrast, on the conceptual or prototype level.

We can evaluate model-driven approaches with two different evaluation goals. First, we can empirically evaluate whether software architects are able to create the necessary input model with sufficient degree of accuracy. Second, we can evaluate how prediction results help in realistic projects in making decisions.

10.1 Empirical Evaluation

We have performed a series of empirical experiments in which we evaluated the applicability of model-driven performance analysis with the PCM [35]. In one of the experiments we conducted, we compared modelling with PCM against modelling with the software performance engineering tool SPE [44].

In the experiment setup two sets of students performed performance analyses of two different systems in a cross-over experiment, i.e., the first group of students used PCM to model the first system and the second group used SPE. For the second system we switched the roles of the groups.

In the experiment, we evaluated both, the time it took the students to model the systems and make performance predictions as well as the correctness of their solution. It turned out that both approaches supported performance predictions to an extend in which the students were able to produce meaningful results. Modelling and analysing with the PCM revealed more issues related to tooling, e.g., issues with the model editors, while modelling with SPE revealed more problems on the conceptual level, especially with non-automated stochastic computations.

10.2 Industrial Case Studies

From the industrial case studies, we report here on a case study performed at IBM and another case study at ABB. We briefly characterise the analysis goal, the analysed system and the main findings. For additional details, please refer to the literature reference of each study. For smaller case studies visit the PCM homepage[1].

IBM Storage Modelling. In our case study performed with IBM Germany [26], we supported the design decision whether to implement the I/O layer of an IBM server system in a synchronous or asynchronous communication style. This layer's main responsibility is to communicate with the hard disks array of the server. Due to external requirements, this layer has to guarantee high throughput. While modelling the system in the PCM showed several obstacles in modelling the system with sufficient degree of accuracy, the results showed that from

[1] http://www.palladio-simulator.com

a performance perspective this decision did not have a significant impact. Both alternatives were comparable in their performance.

ABB Process Control System. In the course of the Q-ImPrESS project we evaluated a process control system at ABB with respect to performance and reliability [30]. The case study revealed good prediction results for performance as well as reliability. However, it also showed that collecting the required input data for the model-driven quality analyses is a non-trivial task and should be better supported in the future. Especially collecting failure rates for reliability models turned out to be a difficult task. Additionally, we also tried to use reverse engineering of models of the ABB PCS system. However, this failed to produce useful models due to the complexity and size of the system's code base.

11 Conclusions and Further Reading

In this article, we give an overview on model-driven analyses methods of non-functional properties. We illustrate a process which consists of the steps modelling the system, annotating the model, transforming the model, solving it, and interpreting the results. For each of the steps, this article gives details of the actions performed by the software architect. Furthermore, we presented brief discussions on practical aspects related to model-driven quality analyses, namely model-driven integration, reverse engineering of the input models from existing systems, and architecture trade-off analyses among several different quality attributes. Case studies show the practicability of the presented approaches in industrial settings.

Software architects use the presented methods to analyse the quality of their systems in various dimensions. They can make dedicated design decisions by taking their quality impacts into account. Also the evolution of existing systems under quality constraints is supported by quality analyses.

In the future, model-driven quality analyses have to deal with systems which are much more dynamic than the systems we modeled in the past. For example, cloud computing results in dynamic allocations where the amount of available hardware changes at run-time. Here, quality analyses at run-time can guide adaptation decisions in order to resolve quality problems of running systems [13]. For this, these approaces use quality models@run-time [9].

References

1. Bause, F.: Queueing petri nets-a formalism for the combined qualitative and quantitative analysis of systems. In: Proceedings of 5th International Workshop on Petri Nets and Performance Models, pp. 14–23 (October 1993)
2. Becker, S.: Coupled Model Transformations. In: WOSP 2008: Proceedings of the 7th International Workshop on Software and Performance, pp. 103–114. ACM, New York (2008)
3. Becker, S.: Coupled Model Transformations for QoS Enabled Component-Based Software Design. Karlsruhe Series on Software Quality, vol. 1. Universitätsverlag Karlsruhe (2008)

4. Becker, S., Dencker, T., Happe, J.: Model-Driven Generation of Performance Prototypes. In: Kounev, S., Gorton, I., Sachs, K. (eds.) SIPEW 2008. LNCS, vol. 5119, pp. 79–98. Springer, Heidelberg (2008), `http://www.springerlink.com/content/62t1277642tt8676/fulltext.pdf`
5. Becker, S., Happe, J., Koziolek, H.: Putting Components into Context: Supporting QoS-Predictions with an explicit Context Model. In: Reussner, R., Szyperski, C., Weck, W. (eds.) Proc. 11th International Workshop on Component Oriented Programming (WCOP 2006), pp. 1–6 (July 2006), `http://research.microsoft.com/ cszypers/events/WCOP2006/WCOP06-Becer.pdf`
6. Becker, S., Hauck, M., Trifu, M., Krogmann, K., Kofron, J.: Reverse Engineering Component Models for Quality Predictions. In: Proceedings of the 14th European Conference on Software Maintenance and Reengineering, European Projects Track, pp. 199–202. IEEE (2010), `http://sdqweb.ipd.kit.edu/publications/pdfs/becker2010a.pdf`
7. Becker, S., Koziolek, H., Reussner, R.: The Palladio component model for model-driven performance prediction. Journal of Systems and Software 82, 3–22 (2009), `http://dx.doi.org/10.1016/j.jss.2008.03.066`
8. Becker, S., Trifu, M., Reussner, R.: Towards Supporting Evolution of Service Oriented Architectures through Quality Impact Prediction. In: 1st International Workshop on Automated Engineering of Autonomous and Run-time Evolving Systems (ARAMIS 2008) (September 2008)
9. Blair, G., Bencomo, N., France, R.: Models@ run.time. Computer 42(10), 22–27 (2009)
10. Bolch, G., Greiner, S., de Meer, H., Trivedi, K.S.: Queueing Networks and Markov Chains. John Wiley & Sons Inc. (1998)
11. Brosch, F., Koziolek, H., Buhnova, B., Reussner, R.: Parameterized Reliability Prediction for Component-Based Software Architectures. In: Heineman, G.T., Kofron, J., Plasil, F. (eds.) QoSA 2010. LNCS, vol. 6093, pp. 36–51. Springer, Heidelberg (2010)
12. Clements, P.C., Bachmann, F., Bass, L., Garlan, D., Ivers, J., Little, R., Nord, R., Stafford, J.: Documenting Software Architectures. SEI Series in Software Engineering. Addison-Wesley (2003)
13. Cortellessa, V., Marco, A.D., Inverardi, P.: Model-Based Software Performance Analysis. Springer, Berlin (2011)
14. Czarnecki, K., Eisenecker, U.W.: Generative Programming. Addison-Wesley, Reading (2000)
15. Czarnecki, K., Helsen, S.: Classification of Model Transformation Approaches. In: OOPSLA 2003 Workshop on Generative Techniques in the Context of Model Driven Architecture (October 2003), `http://www.softmetaware.com/oopsla2003/czarnecki.pdf` (last retrieved January 6, 2008)
16. The DESMO-J Homepage (2007), `http://asi-www.informatik.uni-hamburg.de/desmoj/` (last retrieved January 6, 2008)
17. Eclipse Foundation: xText website, `http://www.xtext.org` (last visited February 22, 2012)
18. Fowler, M., Parsons, R.: Domain Specific Languages. Addison-Wesley, Reading (2010)
19. Gamma, E., Helm, R., Johnson, R., Vlissides, J.: Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley, Reading (1995)

20. Gokhale, S.S.: Architecture-based software reliability analysis: Overview and limitations. IEEE Trans. on Dependable and Secure Computing 4(1), 32–40 (2007)
21. Goldschmidt, T., Becker, S., Uhl, A.: Classification of Concrete Textual Syntax Mapping Approaches. In: Schieferdecker, I., Hartman, A. (eds.) ECMDA-FA 2008. LNCS, vol. 5095, pp. 169–184. Springer, Heidelberg (2008), http://sdqweb.ipd.uka.de/publications/pdfs/goldschmidt2008b.pdf
22. Grassi, V., Mirandola, R., Sabetta, A.: From Design to Analysis Models: a Kernel Language for Performance and Reliability Analysis of Component-based Systems. In: WOSP 2005: Proceedings of the 5th International Workshop on Software and Performance, pp. 25–36. ACM Press, New York (2005)
23. Happe, J., Becker, S., Rathfelder, C., Friedrich, H., Reussner, R.H.: Parametric Performance Completions for Model-Driven Performance Prediction. Performance Evaluation 67(8), 694–716 (2010), http://sdqweb.ipd.uka.de/publications/pdfs/happe2009a.pdf
24. Hermanns, H., Herzog, U., Katoen, J.P.: Process algebra for performance evaluation. Theoretical Computer Science 274(1-2), 43–87 (2002), http://www.sciencedirect.com/science/article/B6V1G-4561J4H-3/2/21516ce76bb2e6adab1ffed4dbe0d24c
25. Hinton, A., Kwiatkowska, M., Norman, G., Parker, D.: PRISM: A Tool for Automatic Verification of Probabilistic Systems. In: Hermanns, H., Palsberg, J. (eds.) TACAS 2006. LNCS, vol. 3920, pp. 441–444. Springer, Heidelberg (2006)
26. Huber, N., Becker, S., Rathfelder, C., Schweflinghaus, J., Reussner, R.: Performance Modeling in Industry: A Case Study on Storage Virtualization. In: ACM/IEEE 32nd International Conference on Software Engineering, Software Engineering in Practice Track, Capetown, South Africa, pp. 1–10. ACM, New York (2010), http://sdqweb.ipd.uka.de/publications/pdfs/hubern2010.pdf
27. Kazman, R., Bass, L., Abowd, G., Webb, M.: SAAM: A method for analyzing the properties of software architectures. In: Fadini, B. (ed.) Proceedings of the 16th International Conference on Software Engineering, pp. 81–90. IEEE Computer Society Press, Sorrento (1994)
28. Koziolek, H.: Parameter Dependencies for Reusable Performance Specifications of Software Components. The Karlsruhe Series on Software Design and Quality, vol. 2. Universitätsverlag Karlsruhe (2008)
29. Koziolek, H., Reussner, R.: A Model Transformation from the Palladio Component Model to Layered Queueing Networks. In: Kounev, S., Gorton, I., Sachs, K. (eds.) SIPEW 2008. LNCS, vol. 5119, pp. 58–78. Springer, Heidelberg (2008), http://www.springerlink.com/content/w14m0g520u675x10/fulltext.pdf
30. Koziolek, H., Schlich, B., Bilich, C., Weiss, R., Becker, S., Krogmann, K., Trifu, M., Mirandola, R., Koziolek, A.: An industrial case study on quality impact prediction for evolving service-oriented software. In: Proceeding of the 33rd International Conference on Software Engineering, Software Engineering in Practice Track, ICSE 2011, pp. 776–785. ACM, New York (2011), http://doi.acm.org/10.1145/1985793.1985902
31. Krogmann, K., Kuperberg, M., Reussner, R.: Using Genetic Search for Reverse Engineering of Parametric Behaviour Models for Performance Prediction. IEEE Transactions on Software Engineering 36(6), 865–877 (2010), http://sdqweb.ipd.kit.edu/publications/pdfs/krogmann2009c.pdf
32. L'Ecuyer, P., Buist, E.: Simulation in Java with SSJ. In: WSC 2005: Proceedings of the 37th Conference on Winter Simulation, Winter Simulation Conference, pp. 611–620 (2005)

33. Malavolta, I., Muccini, H., Pelliccione, P., Tamburri, D.A.: Providing architectural languages and tools interoperability through model transformation technologies. IEEE Transactions of Software Engineering 36(1), 119–140 (2010)
34. Marshall, R.: SAP gives update on Business ByDesign plans (2009), http://www.v3.co.uk/v3-uk/news/1970547/sap-update-business-bydesign-plans (last visited November 22, 2009)
35. Martens, A., Koziolek, H., Prechelt, L., Reussner, R.: From monolithic to component-based performance evaluation of software architectures. Empirical Software Engineering 16(5), 587–622 (2011), http://dx.doi.org/10.1007/s10664-010-9142-8
36. Meier, P., Kounev, S., Koziolek, H.: Automated Transformation of Palladio Component Models to Queueing Petri Nets. In: 19th IEEE/ACM International Symposium on Modeling, Analysis and Simulation of Computer and Telecommunication Systems (MASCOTS 2011), Singapore, July 25-27 (2011)
37. Object Management Group (OMG): MOF 2.0 Core Specification (formal/2006-01-01) (2006), http://www.omg.org/cgi-bin/doc?formal/2006-01-01
38. Object Management Group (OMG): Meta Object Facility (MOF) 2.0 Query/View/Transformation Specification – Version 1.1 Beta 2 (December 2009), http://www.omg.org/spec/QVT/1.1/Beta2/
39. Object Management Group (OMG): UML Profile for MARTE: Modeling and Analysis of Real-Time Embedded Systems, version 1.0 (2009), http://www.omg.org/spec/MARTE/1.0/PDF
40. Parr, T.: The Definitive ANTLR Reference Guide: Building Domain-specific Languages (Pragmatic Programmers). Pragmatic Programmer (2007)
41. Petriu, D.B., Woodside, M.: An intermediate metamodel with scenarios and resources for generating performance models from UML designs. Software and Systems Modeling 6(2), 163–184 (2007)
42. Rolia, J.A., Sevcik, K.C.: The Method of Layers. IEEE Transactions on Software Engineering 21(8), 689–700 (1995)
43. Saaty, T.L.: The Analytic Hierarchy Process, Planning, Piority Setting, Resource Allocation. McGraw-Hill, New York (1980)
44. Smith, C.U., Williams, L.G.: Performance Solutions: A Practical Guide to Creating Responsive, Scalable Software. Addison-Wesley (2002)
45. Völter, M., Stahl, T.: Model-Driven Software Development. Wiley & Sons, New York (2006)
46. Woodside, M., Petriu, D.C., Siddiqui, K.H.: Performance-related Completions for Software Specifications. In: Proceedings of the 22nd International Conference on Software Engineering, ICSE 2002, Orlando, Florida, USA, May 19-25, pp. 22–32. ACM (2002)