Rapid Performance Modeling by Transforming Use Case Maps to Palladio Component Models

Christian Vogel*, Heiko Koziolek†, Thomas Goldschmidt†, Erik Burger*
*Software Design & Quality, Karlsruhe Institute of Technology, Germany
E-Mail: sdq@ipd.uka.de,burger@kit.edu
†ABB Corporate Research, Industrial Software Systems Program, Ladenburg, Germany
E-Mail: {heiko.koziolek,thomas.goldschmidt}@de.abb.com

ABSTRACT

Complex information flows in the domain of industrial software systems complicate the creation of performance models to validate the challenging performance requirements. Performance models using annotated UML diagrams or mathematical notations are difficult to discuss with stakeholders from the industrial automation domain, who often have a limited software engineering background. We introduce a novel model transformation from Use Case Maps (UCM) to the Palladio Component Model (PCM), which enables performance modeling based on an intuitive notation for complex information flows. The resulting models can be solved using existing simulators or analytical solvers. We validated the correctness of the transformation with three case study models, and performed a user study. The results showed a performance prediction deviation of less than 10 percent compared to a reference model in most cases.

Categories and Subject Descriptors

D.2.8 [**Software Engineering**]: Metrics—*complexity measures, performance measures*; D.2.11 [**Software Engineering**]: Software Architecture

General Terms: Design, Performance, Requirements Engineering

1. INTRODUCTION

Industrial software systems, such as distributed control systems, follow complex information flows. For example, they transport sensor data from industrial field devices along multiple hubs to central servers, from which the data can be processed by different kinds of operator stations. Designing software architectures for such systems is hard because of the complex control flows. In addition, these systems have challenging performance requirements for high throughput and short response times. Thus, early performance modeling is desired to identify performance bottlenecks and design efficient information flows.

To enable early performance modeling, numerous mathematical notations (e.g., queuing networks, stochastic Petri nets, stochastic process algebra) and software-related notations (e.g., annotated UML) have been proposed [1, 29, 12]. These notations usually require expertise from the performance domain and their graphical representations are often difficult to discuss with stakeholders from the domain of industrial systems. This complicates reasoning on different alternatives for scalable architectures and leads to a general reluctance to apply performance modeling in this domain.

The notation of Use Case Maps (UCM) [11] has been created by analyzing how software systems are typically sketched in early stages on white boards. UCMs capture high level software components as well as the control flow for specific usage scenarios in an intuitive notation. In former work, Petriu and Woodside [20] proposed a model transformation from UCMs to layered queuing networks (LQN) to allow performance predictions from UCMs. Their transformation did however not result in a component-based model, and the implementation is no longer maintained. Other transformations mapped UML state machines, sequence diagrams, or activity diagrams to stochastic Petri nets [3] or stochastic process algebras [26], but require UML/MARTE [18] and/or performance expertise to be useful.

The contribution of this paper is UCM2PCM, a model transformation from UCMs into the Palladio Component Model (PCM) [2], which enables component-oriented performance modeling as well as multiple solution methods (e.g., discrete event-based simulation [2], layered queuing networks [13], queuing Petri nets [15]). The transformation bridges semantic gaps between the requirements-oriented UCM notation and the component-oriented PCM notation. Users can create UCMs using existing graphical editors and transparently run performance solvers using the PCM tool chain. Our approach supports a two-staged software performance engineering process, where first domain stakeholders can perform initial modeling based on UCMs and second performance experts and architects can refine the resulting PCM models.

In this paper, we evaluate UCM2PCM by transforming UCMs from three existing software systems into respective PCM models. We validate the correct bridging of seman-

ICPE '13, April 21–24, 2013, Prague, Czech Republic.

http://dx.doi.org/10.1145/2479871.2479888

tical gaps between the notations by comparing UCM-based simulation results with former purely PCM-based simulation results. In addition, we validate the usability of the models and tooling with an empirical user survey, demonstrating the benefits of the new modeling approach. UCM2PCM [27] is available for free, allows faster modeling, and requires a lower effort for model modifications, thus increasing the motivation to try different design alternatives. UCMs provide a better overview of systems with complex control flows and can be easier discussed with domain stakeholders.

The remainder of this paper is organized as follows: Section 2 briefly introduces UCM and PCM. The mapping from UCMs to PCM is described in Section 3. Section 4 presents the evaluation of the approach, including the results of the survey. Section 5 differentiates our approach from related work. The last Section 6 provides conclusions and future work.

2. FOUNDATIONS

This section provides the backgrounds of this work and introduces the Use Case Maps language and the Palladio Component Model.

2.1 Use Case Maps (UCM)

As part of the User Requirements Notation (URN) specification [11], UCMs are used to visualize how a system works and what the requirements and causal responsibilities are. UCMs are behavioral diagrams and use scenarios. This is similar to UML sequence diagrams, but UCMs remain on a higher abstraction level. In difference to sequence diagrams, UCMs don't show all messages or signals that are exchanged between components or actors, but only control flows with importance for the behavior of a system. Skipping the details enhances the overview and allows the usage of UCM early in the design process, where not much detail is specified yet [5].

The design of UCMs is clear and simple and the notation is intuitive because it is based on sketching a model by hand on a piece of paper. This shall make UCM easy to learn and understandable, also for non-experts.

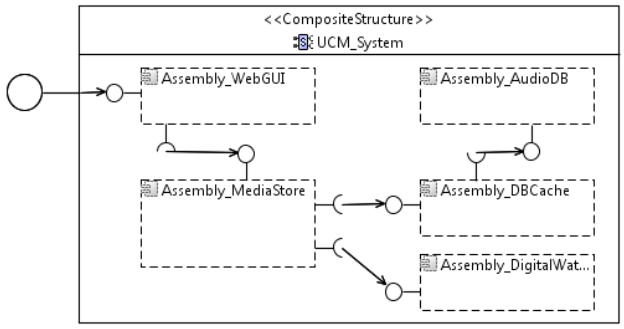
A UCM diagram consists of paths, showing a possible control flow through a system. In Fig. 1 an example of a UCM diagram for a software implementing a media store can be found. Every path has at least one `Start-` and one `End-Point`. Along each path so called `Responsibilities` describe the actions that take place on a high abstraction level. `Responsibilities` are symbolized with a cross. Alternatively, `Stubs` can be used. A `Stub` is displayed as a rhombus and represents an own UCM diagram that describes in more details what happens inside.

To model the control flow, forks can be added to a path. An `AndFork` splits a path into two branches, which are executed in parallel. An `OrFork` offers alternative branches. To decide, which branch is taken, conditions or probabilities can be specified. The `OrJoin` joins branches. Additionally, an `AndJoin` contains a barrier, which continues the control flow only when all incoming branches finished executing.

UCMs also support the modeling of components. With components, the entities involved in a scenario can be specified and an architectural structure of a system can be defined. Components in UCMs can also be nested. In a diagram, components are represented by boxes. All elements inside the box are bound to that component. Unlike in the PCM, there are no interface definitions for components. Their external behavior can only be derived by the paths entering or exiting the component. To model different design alternatives with UCM, it is often sufficient to keep the paths and only replace or rearrange the components of a model.

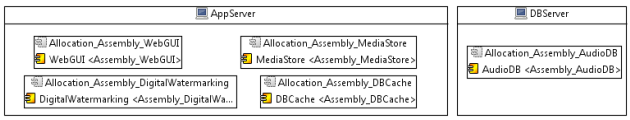
2.2 Palladio Component Model (PCM)

The PCM [2] is a domain-specific tool for component based software engineering (CBSE) that is based on the Eclipse Modeling Framework (EMF). Besides other analyses it supports performance prediction for software models [24]. It offers the comparison and selection of several design alternatives of a software system already in the modeling state. The PCM has recently been applied in a number of industrial case studies [14, 8, 23].



(a) PCM System of the Media Store

(b) Example PCM SEFF of the Media Store's WebGUI Download Service



(c) PCM Allocation of the Media Store

Figure 2: Media Store PCM Example

A PCM model consists of four parts that cover different aspects of the performance model:

- **Repository:** The repository contains the component types & interfaces of the system. There are `Basic-` and `CompositeComponents`. `BasicComponents` contain so called `ServiceEffectSpecification (SEFF)` that abstractly describe performance-relevant behavior of a single component service. `CompositeComponents` are composed from inner components. A `SEFF` contains actions that determine the control flow inside the component. For example, Fig. 2b depicts a `SEFF`

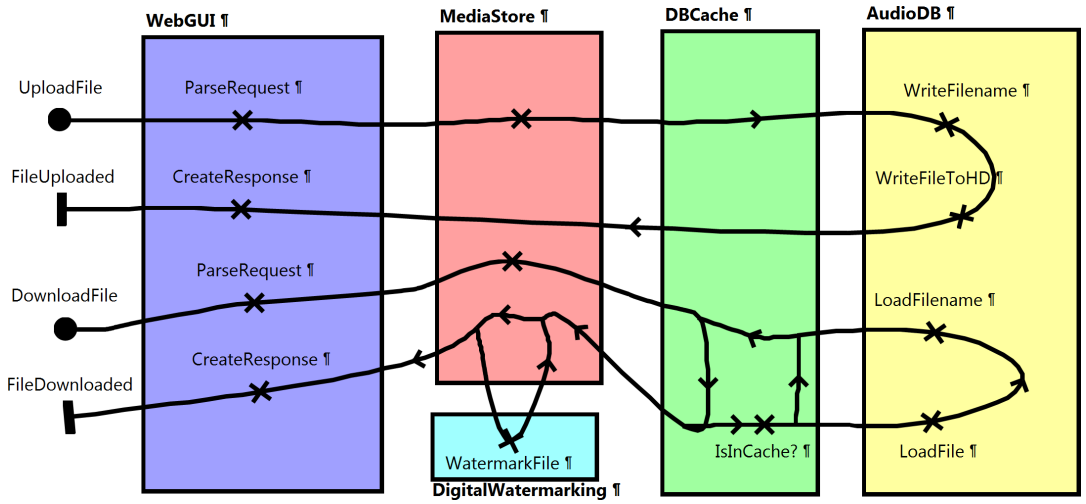


Figure 1: UCM of the Media Store components and inner control flow. The usage model is another UCM belonging to the model, which is omitted here for space reasons.

for the MediaStore's WebGUI service 'download'. Every `SEFF` starts with a `StartAction` and ends with an `EndAction`. `InternalActions` represent computations or other resource consumption inside a component and thus contain execution time annotations that are later transformed into service demands and used by the performance simulator. `ExternalCallActions` can be used to call external services that are required. `Fork-`, `Branch-` and `LoopActions` are used to manage the control flow. Components can provide or require interfaces and are bound to them via `Provided-` or `RequiredRoles`.

- **System:** A system is a composite structure and shows how the components in a software system are connected (cf. example in Fig. 2a). It consists of `AssemblyContexts` that represent instances of the components in the repository and thus having the same `Provided-` or `RequiredRoles`. If the `Provided-` and `RequiredRole` of two `AssemblyContexts` match, they can be connected by an `AssemblyConnector`. Like every composite structure, a system itself can also have `Provided-` or `RequiredRoles`. A `SystemProvidedRole` represents services that the software system provides to its environment. With a `ProvidedDelegationConnector` the `SystemProvidedRole` is connected to the `ProvidedRole` of the inner `AssemblyContext` that offers the provided service. If an `AssemblyContext` needs to call a service outside the system, its `RequiredRole` is connected to the `SystemRequiredRole`, using a `RequiredDelegationConnector`. By that, the request for a service is forwarded to the system boundaries.

- **Allocation:** The allocation defines, how component instances (i.e., `AssemblyContexts`) are deployed on hardware resources (cf. example in Fig. 2c). The mapping relation for a component instance is called `AllocationContext`. The hardware resources are stored in a `ResourceEnvironment`. Each hardware resource can have different processing entities, such as CPU or HDD.

- **Usage Model:** In a usage model, several `UsageScenarios` can be created, each describing an expected usage of the system, by a user. For each `UsageScenario` a workload (i.e., open or closed) and a behavior can be defined. This behavior consists of a set of `EntryLevelSystemCalls` that refer to services in `SystemProvidedRoles`. In this set, also sequences, branches and loops can be used.

The PCM Workbench provides multiple tools to conduct the performance predictions. The tool PCM Solver can be used to calculate the execution times and the related probabilities of a workload. The second tool, SimuBench allows simulating different number of runs for a workload. For every call in the model, the execution times can be simulated and visualized in several diagrams. Additionally, diagrams that show the utilization of hardware resources, such as CPUs or hard disks, can be created.

3. UCM TO PCM MAPPING

This section describes how UCM elements or constructs are translated into corresponding PCM elements or constructs, which performance relevant annotations to UCM elements are specified and what limitations exist for the transformation. The main idea of the transformation is to map UCM components to PCM components and then to segment each UCM path per component to create PCM SEFFs.

3.1 Paths

Every path in a UCM corresponds with a `SystemProvidedRole`, as both represent a function that the software system provides to its environment (Fig. 3). In contrast to standard UCMs, UCM paths to be transformed into PCM need exactly one `StartPoint` and one `EndPoint`. Different paths may not join. Otherwise, the mapping could be ambiguous in the UCM2PCM tool. To translate a path into PCM, the UCM2PCM transformation follows a path node by node. In the following, the mapping for the most important UCM constructs is presented.

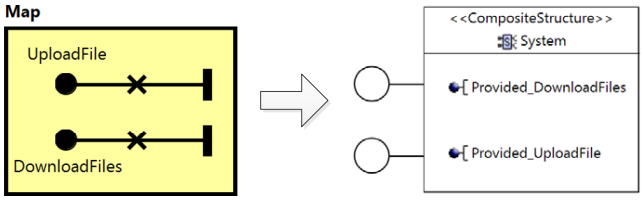


Figure 3: UCM2PCM Mappings - UCM Map

3.2 Hardware resources

To be able to predict the performance of a component, it has to be mapped to an active resource. This is done in PCM with an `AllocationContext`. In UCMs, there is no corresponding construct. Therefore, a performance annotation in the meta data of the UCM component is needed. Here, the allocation to a hardware resource can be specified as a key-value pair as shown in Fig. 4. If the allocation meta data is missing, then the component is allocated to a default hardware resource.

If a PCM hardware resource is not yet present from former transformation steps, the transformation automatically creates it first. All hardware resources are created with a default CPU and a default HDD as processing entities, no parameters can be specified here. This is a deliberate limitation to keep modeling with the UCM2PCM tool simple and allow for rapid performance modeling. However, if servers with custom CPU or HDD are needed, then this can be defined later in PCM after the transformation as manual refinement step.

3.3 Components

In a UCM, components can be reused in different diagrams. That is why the component *definitions* do not appear in a UCM diagram. The boxes in a diagram only are references to UCM components defined elsewhere. This is the same concept as in PCM, where a component of the PCM repository can be reused in several `AssemblyContexts` in a software system. In the transformation therefore, for every reference an own `AssemblyContext` will be created. If the corresponding component already exists in the PCM repository, it will be used; otherwise a new component for that `AssemblyContext` will be created.

In UCM, all component references that are based on the same component share the same name and meta data. These attributes are stored in the component and not in its references. This means that the corresponding `AssemblyContexts`, created by the transformation, also share the same name and are allocated on the same hardware resource. If two `AssemblyContexts`, based on the same component, should be named differently or they should be allocated on different hardware resources, this has be to done manually in the PCM model resulting from the UCM2PCM transformation. We decided that this limitation is acceptable, because to solve this problem the UCM meta model would have to be changed.

Fig. 4 shows, how a normal UCM component reference is translated into a `BasicComponent`. Additionally, a new `SEFF` with a `StartAction` will be created for the component, to

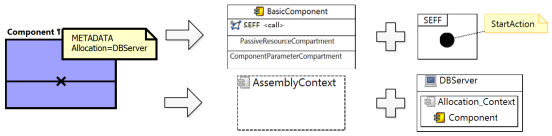


Figure 4: UCM2PCM Mappings - Standard Component

follow the path inside the component. If the UCM component reference contains a `Stub`, then a `CompositeComponent` will be created instead in the PCM repository. This can be seen in Fig. 5.

3.4 Composite Components

`Stubs` can be used to create layered UCMs. They contain a complete UCM map. Every path leading inside or outside a `Stub` can be connected to a `Start-` or `EndPoint` inside the Stub. `CompositeComponents` have a similar function in PCM. They are composed from inner components and can be seen as a kind of subsystem. Therefore, a direct mapping between a UCM component containing a `Stub` and a `CompositeComponent` can be defined. A `CompositeComponent` itself may not contain any actions. This is only possible for the inner components. Therefore, in a UCM component containing a `Stub`, also no additional nodes are allowed; otherwise, no mapping is possible and the component can't be translated into PCM.

For all components inside a `Stub`, the allocation meta data will be ignored, because all components inside a `CompositeComponent` in PCM are defined to be allocated on the same hardware resource as the `CompositeComponent`.

In PCM, actions can only occur inside components, while in UCMs there are no dependencies between nodes and components. To be able to map a UCM path into PCM, the UCM2PCM transformation ignores all nodes or constructs, which are located outside a component, except for `Responsibilities` that are mapped as `SystemRequiredCalls`.

By moving a node inside or outside a component, it can quickly be included or excluded from the transformation. This can even accelerate the process of testing and converting different design alternatives into PCM.

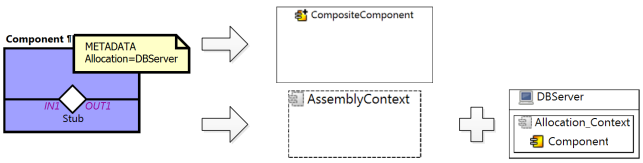


Figure 5: UCM2PCM Mappings - Stub Component

3.5 Responsibilities and Resource Demands

For UCM `Responsibilities`, there is a 1:1 mapping with `InternalActions`, where resources are consumed. Therefore, resource demands performance annotations have to be specified for `Responsibilities`. The UCM2PCM tool supports the CPU & HDD resources of PCM. They can be entered as key-value pairs in the meta data of the `Responsibilities` as shown in Fig. 6. If no resource demand is specified, a default CPU demand will be used.

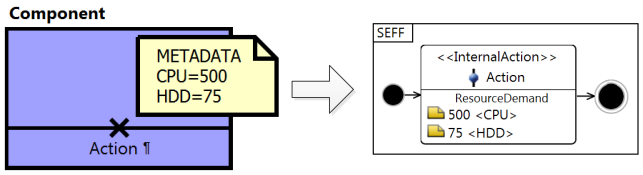


Figure 6: UCM2PCM Mappings - Responsibilities

In PCM models, it can be necessary to model a control flow that enters a component without any `InternalAction` executed. For example if a component forwards a call to another component, instead of processing it internally. In UCMs, this behavior is not possible. If a new component is entered, there always has to be a node inside the component, otherwise it could not be determined if the component has been entered. As a workaround, a `Responsibility` with an explicit CPU resource demand of zero can be used for this purpose. The transformation skips all `Responsibilities` with CPU resource demand of zero.

3.6 Calls

Component calls to other components have no dedicated representation in a UCM but they implicitly occur when the path enters or leaves a component. In UCMs, a call can be determined, by checking if the surrounding component of a node has changed compared to the predecessor node.

PCM implements the concept of interfaces as a contract between different components. Therefore, an interface has to be created for every call in a UCM, by the transformation. In general, a `RequiredRole` targeting this interface has to be created for the caller, and a `ProvidedRole` has to be created for the callee. An example of this is shown in Fig. 7.

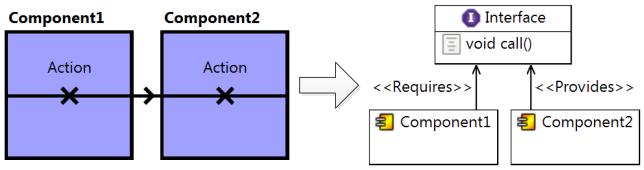


Figure 7: UCM2PCM Mappings - Call Interfaces

The transformation can distinguish different types of calls, depending on whether a component has been entered, been changed, been left or if a Stub has been entered or left. The following enumeration describes the different calls in more detail:

1. **SystemProvidedCall**: Here, the previous node is a `StartPoint` and the path has entered a new UCM component as depicted in Fig. 8. In this case, for the new component & `AssemblyContext` in PCM, a `ProvidedRole` corresponding to the call is created. In addition the `ProvidedRole` will be cloned and added as a `SystemProvidedRole` to the `AssemblyContexts` parent container. This container is the PCM system, or a `CompositeComponent`, if the path resides inside a `Stub`. Then the two `ProvidedRoles` will be connected by a `ProvidedDelegationConnector`.

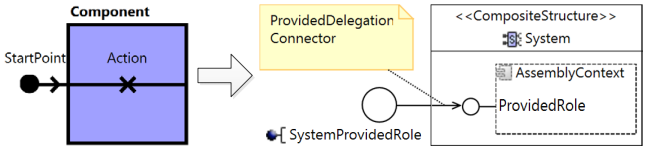


Figure 8: UCM2PCM Mappings - SystemProvidedCall

2. **ComponentCall**: A `ComponentCall` occurs when a UCM path crosses the borders between two UCM components. In this case the component of the current node is different than the component of its predecessor node, as shown in Fig. 9. For the component, which is left a `RequiredRole`, for the component, which is entered a `ProvidedRole`, corresponding to the call, are created. Additionally, the transformation creates an `AssemblyConnector` that connects the created `Provided-` and `RequiredRole`. Finally, in the `SEFF` of the calling component, an `ExternalCallAction` targeting the new `RequiredRole`, is added.

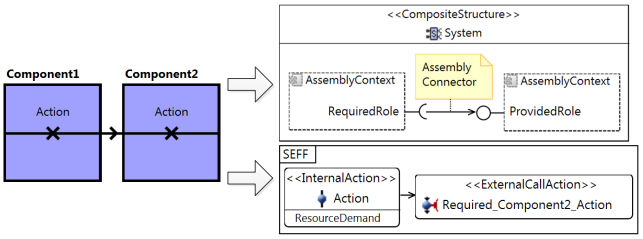


Figure 9: UCM2PCM Mappings - ComponentCall

3. **SystemRequiredCall**: A `SystemRequiredCall` is used, if a component needs to be called, which is outside the modeled software system. This is depicted in Fig. 10. In a UCM, such a call can be modeled by a path leaving a component, followed by a `Responsibility` outside of a component. In the transformation, a new `RequiredRole` for the calling PCM component is created. In addition, the `RequiredRole` is cloned and added to the PCM System as `SystemRequiredRole`. Finally, both `RequiredRoles` will be connected, using a `RequiredDelegationConnector`. In the `SEFF` of the calling component, an `ExternalCallAction` targeting the new `RequiredRole` is added. Inside a `Stub`, a `SystemRequiredCall` is not possible. Here, a `ParentCall` has to be used instead. A `SystemRequiredCall` is the only case, where a node outside a component is used to create PCM elements. In all other cases, nodes outside a component will be ignored by the UCM2PCM transformation.

4. **ParentCall**: A `ParentCall` is used, inside a `Stub`, to call a target, which is outside the `Stub`. This is equivalent to a `SystemRequiredCall`, as in both cases the target of the call is outside the parent container. The syntax of a `ParentCall` is shown in Fig. 11. Inside a `Stub`, there has to be an `EndPoint` inside the calling component. In the same component there also needs to be a `StartPoint`, where the control flow returns back into the `Stub`. Both the `End-` and the `StartPoint` need to be connected by a path leaving and entering

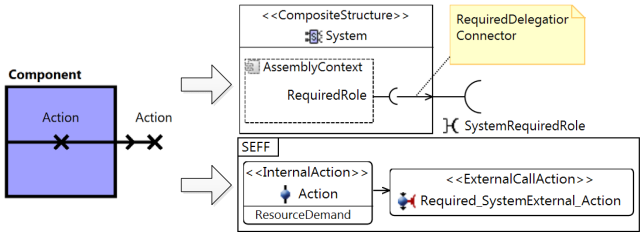


Figure 10: UCM2PCM Mappings - SystemRequiredCall

the `Stub`. In the path outside the `Stub`, any types of calls can be made. The `ParentCall` will be translated similar as the `SystemRequiredCall`. A `RequiredRole` and a `SystemRequiredRole` are created and connected via a `RequiredDelegationConnector`.

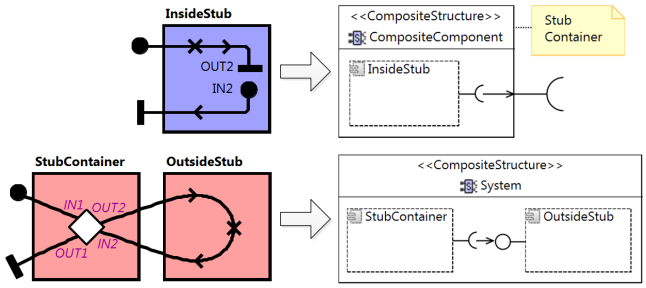


Figure 11: UCM2PCM Mappings - ParentCall

In a PCM model, `ExternalCallActions` model synchronous calls by default. Therefore, all calls in UCMs are also assumed to be synchronous. If there is an open call, meaning that the control flow has not yet returned to the caller, and the calling component is reentered, then the transformation interprets this as the return of the open call, rather than a new call. Before the call returns, all calls opened in the meantime will automatically return and get closed in the opposite order than they were opened. When a path reaches an `EndPoint` and is finished, all calls, which are still open, are automatically returned and closed.

However, if it is necessary to model components in UCMs that are calling each other with circular dependency (e.g. in the callback software pattern), then this could be realized using a workaround. To avoid a call being interpreted as return call, an `OrFork` can be entered before the call. The call needs to be moved to one branch of the `OrFork` and its probability has to be set to '1'. The probability of the other branch will be set to '0'. The left diagram in Fig. 12 shows the default behavior, while the workaround is shown on the right side.

3.7 Branches

Beside the sequential control flow, UCM2PCM also supports modeling all branching constructs of the PCM. In PCM there are `Fork-`, `Branch-` and `LoopActions`. In UCMs, these actions can be modeled using `AndForks`, `AndJoins`, `OrForks` and `OrJoins`.

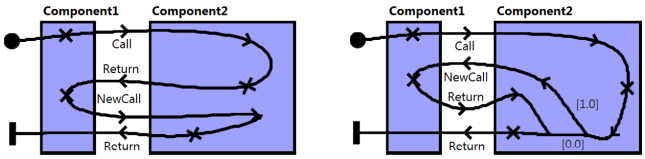


Figure 12: UCM2PCM Mappings - Workaround for Components Calling Each Other

1. A `ForkAction` in PCM consists of several `ForkedBehaviors` that contain subpaths, which are executed in parallel. In UCM, an `AndFork` with corresponding `AndJoin` has the same semantic meaning, therefore there is a 1:1 mapping of both constructs, depicted in Fig. 13.

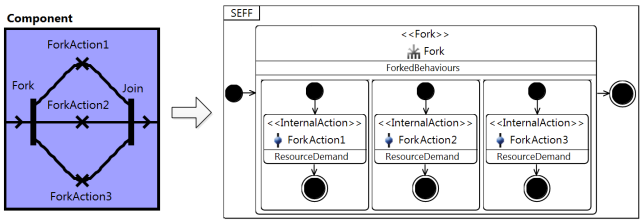


Figure 13: UCM2PCM Mappings - Forks

2. A `BranchAction` in PCM also splits the control flow, but, in contrast to a `ForkAction`, only one of the possible paths will be followed. In PCM, there are guarded and probabilistic `BranchActions`. In guarded `BranchActions`, the choice of which branch is taken is depends on guard conditions. They are not supported by the UCM2PCM transformation. For probabilistic `BranchActions`, probabilities have to be specified for each branch. In UCMs, they can be modeled by an `OrFork` with corresponding `OrJoin`, as seen in Fig. 14. The probabilities of each branch in UCM have to be specified in the attribute "probability" of the connection, which connects the `OrFork` with the first node in the subpath in the UCM diagram.

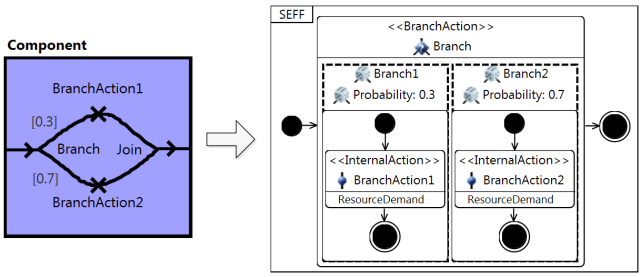


Figure 14: UCM2PCM Mappings - Branches

3. `LoopActions` can only be emulated in UCMs. There is no special loop construct to model them 1:1. For the UCM2PCM transformation, a loop is defined as an `OrJoin`, directly followed by an `OrFork` with two branches. One branch, containing the loop body, leads back to the `OrJoin`, the other branch continues the path after the loop. From the performance point of

view the number of iterations for the loop is relevant. This information can be specified as performance annotation, in the meta data of the `OrFork`, as shown in Fig. 15.

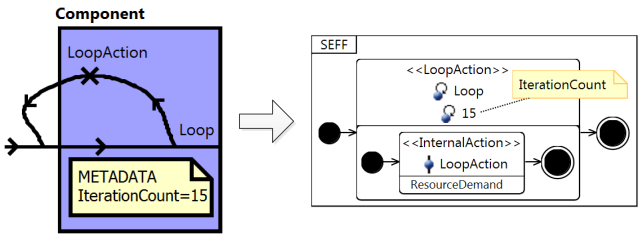


Figure 15: UCM2PCM Mappings - Loops

In the context of the UCM2PCM transformation, the usage of join nodes is affiliated with two problems. Firstly, the fork and the corresponding join node could be located in different components, which is not supported by PCM. To avoid this problem, we decided that the fork node in UCM defines, in which component the resulting action in PCM will be located. The join node is automatically assumed to be located in the same component.

Secondly, there is a problem with nesting. Like in PCM, all branching constructs can be nested. The use of joins in UCM can lead to a "dangling join" problem. Therefore, the transformation is restricted to only hierarchical nesting. This implies that each join, which is not part of a loop, belongs to the latest opened fitting fork. The UCM2PCM transformation contains a special algorithm [27] to check, if the branching in a UCM is valid.

3.8 UsageScenarios

The built-in rudimentary scenario support in UCMs does not match the capabilities of PCM. Therefore, we decided that `UsageScenarios` have to be modeled explicitly with UCMs. This is possible with hardly any additional effort and allows us to also model sequential `EntryLevelSystemCalls` with branches and loops.

A UCM component of type "Actor" represents a `UsageScenario`, as depicted in Fig. 16. In the meta data, the workload can be specified with the same properties as in PCM. If the workload is open, the interarrival time of new users has to be specified; if it is closed, then the population and the think time are necessary parameters.

The whole path of the scenario including the `StartPoint` has to be located inside the "Actor" component. This allows the transformation to distinguish between a normal path through the software system and a `UsageScenario`. Branches and Loops can be built like in a normal UCM path. The `Responsibilities` inside a scenario are translated into `EntryLevelSystemCalls`, instead of `InternalActions`. Therefore their name has to be the same as the name of the `StartPoint` representing the `SystemProvidedRole` that should be called by the `EntryLevelSystemCall`.

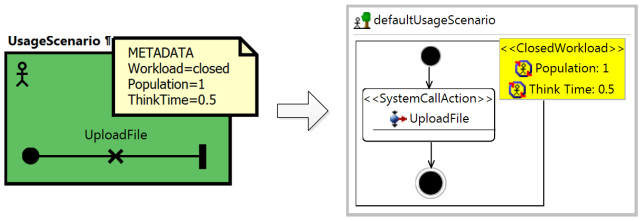


Figure 16: UCM2PCM Mappings - UsageScenarios

3.9 Assumptions and Limitations

There are a few limitations for the use of the UCM2PCM tool that are described in the following:

- **Limited PCM feature support:** Our approach is not intended to fully support all PCM features; instead, we propose a trade-off between ease-of use and complexity of a tool. Thus, PCM variables and guarded branches are not supported. Right now, only fixed values or the PCM "Stochastic Expressions" [24] can be used for performance annotations in UCMs. It is also not possible to create a custom `ResourceEnvironment`. Allocating two `AssemblyContexts`, implementing the same component, onto different hardware resources is infeasible, due to UCM meta model limitations.
- **Limited input assistance:** The jUCMNav editor for creating UCMs [22] has limited input assistance. Performance annotations need to be added in the meta data of each UCM node, as key-value pairs. This is not convenient, as the user needs to know the names of all performance annotations. At design time, in the UCM editor, there is no possibility to validate a UCM model and to indicate errors to a user. This is only possible after modeling, by running the UCM2PCM transformation.
- **No reverse transformation:** After transforming a UCM into a PCM model, this model can be further refined in PCM. Thus, there is no reverse transformation yet, to transfer the changes back into the UCM. However, it would be desirable to regain an easy presentable UCM representation of the refined performance model.

3.10 Implementation

Fig. 17 shows a high-level sketch of the UCM2PCM tool chain. The UCM2PCM prototype is implemented as an Eclipse plug-in. The transformation uses the Eclipse Model to Model (M2M) transformation framework and is implemented in QVT Operational [17] (approx. 3000 lines of code). UCMs can be created with the jUCMNav editor [22] based on the Graphical Modeling Framework (GMF). UCM2PCM can then be invoked from inside Eclipse accepting a UCM model as input. The resulting PCM model can be further refined by editing it with the PCM Workbench [6], which is also based on GMF.

The transformations from the resulting PCM model into the different performance models are out of the scope of this paper. Becker et al. [2] describe the transformation of PCM

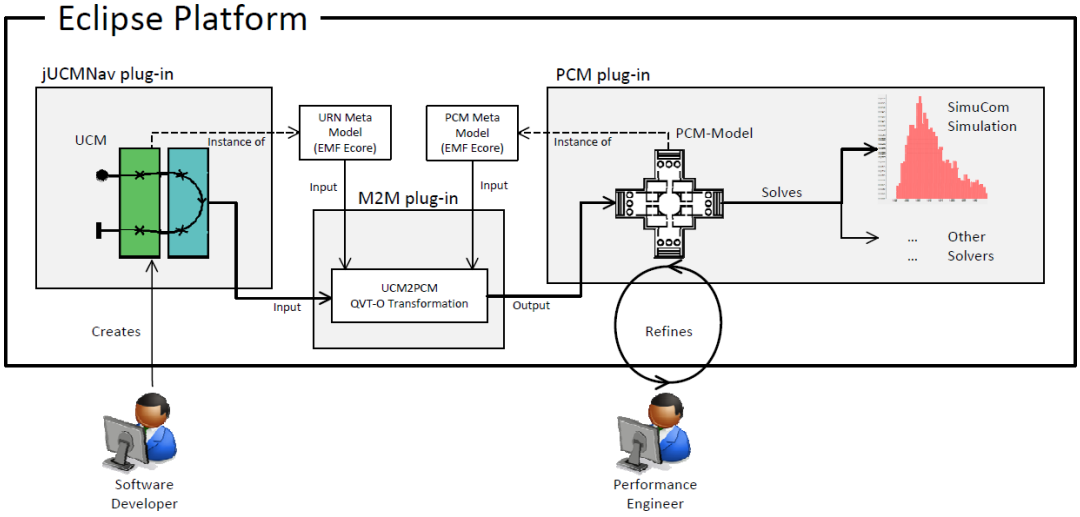


Figure 17: UCM2PCM Toolchain

instances into extended queuing networks, which are then solved using the discrete-event simulator SimuCom. Koziolek et al. [13] map PCM instances to layered queuing networks, which are then solved using the fast, heuristic layered queueing network solver. Meier et al. [15] explain how to map PCM instances to queueing Petri nets, which are solved by approximate analytical solvers or simulation.

The whole tool chain is open source and available free of charge.

4. EVALUATION

To validate UCM2PCM, we need to show that its mapping of model elements is valid and the tool is useful. Thus, this section evaluates (a) the accuracy of the transformation (i.e., the correct mapping) and (b) the usability of the UCM model. To address (a), we first ran a number of internal tests with a number of artificial models which are not detailed here. Afterwards, we transformed performance-annotated UCMs of three different case study systems into the respective PCM models (Section 4.1). We then ran a series of experiments (Section 4.2) and compared the simulation results of these models with simulation results from former PCM models of these systems (Section 4.3). A low difference between both simulation results demonstrates that UCM2PCM successfully bridges semantic gaps between UCM and PCM. To address (b), we let six users model a UCM and apply the UCM2PCM transformation (Section 4.4). We then asked them for feedback in a user survey. While the low number of users is not statistically relevant to draw general conclusion, it provides a first hint on the usability of the approach.

4.1 Systems under Study

We applied UCM2PCM on three heterogeneous, mid-sized systems, to increase the external validity of our approach. The models also demonstrate that UCM2PCM can be used to model and process complex control flows and is therefore well-suited for industrial control systems. The analyzed models are:

- **Media Store:** This system (Fig. 1) is a plain Java web application for storing and retrieving audio or video files using a MySQL database. The model reflects a use case where a digital watermark is added to downloaded files for copy protection. The model contains a hard disk resource, which is accessed when retrieving files. Resource demands for the Media Store have been measured using manual instrumentation of the Java implementation [2]. The model comprises 19 resource demands and a closed workload.
- **SPECjAppServer:** SPECjAppServer is an industry-standard benchmark, designed to measure the performance of application servers conforming to the Java EE 5.0 or later specifications. It is modeled after an automobile manufacturer, where dealers place customer orders or interact with suppliers. The system is implemented as a Java Enterprise application, deployed on two servers using an Oracle database. The 15 resource demands in the model have been determined using an estimation technique based on measured response times and resource utilization [4].
- **Business Reporting System:** The BRS (Fig. 18) is loosely modeled after a management information system, formerly analyzed at Carlton University [30]. Users can retrieve live business data from the system and run statistical analyses. The analyzed configuration comprises two open workload usage scenarios, nine components, and four servers. The 37 resource demands of the BRS are based on estimations. Fig. 18 depicts a high-level overview of five of the most important UCM paths in the model. The model includes UCM `Stubs` with nested UCMs that are mapped to PCM `CompositeComponents`. Fig. 19 shows the PCM repository resulting from the UCM2PCM transformation.

4.2 Experiments

Based on the available PCM instances of the systems, we manually created corresponding UCMs, using jUCMNav 4.3.0. We had to resolve parameter dependencies from

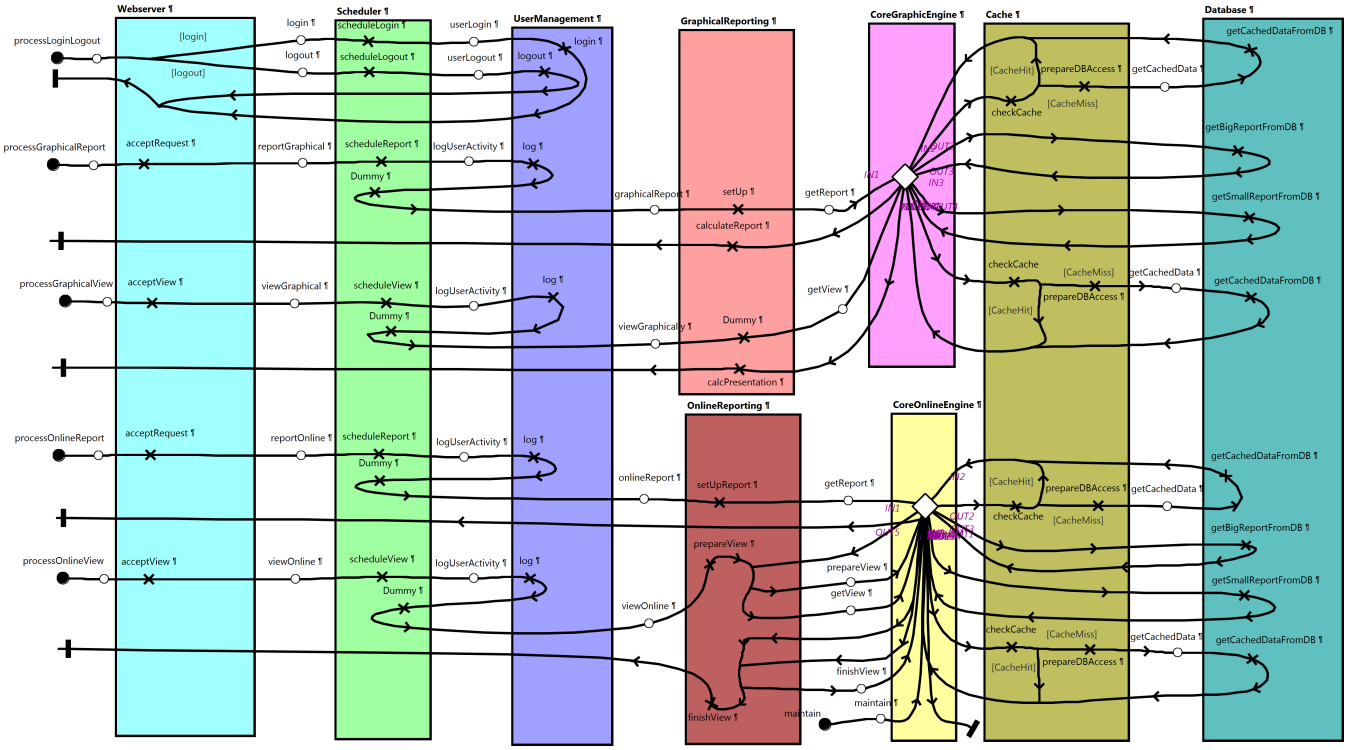


Figure 18: UCM of the high-level Business Reporting System. Nested UCM Stubs and the corresponding UCM usage model are not shown here.

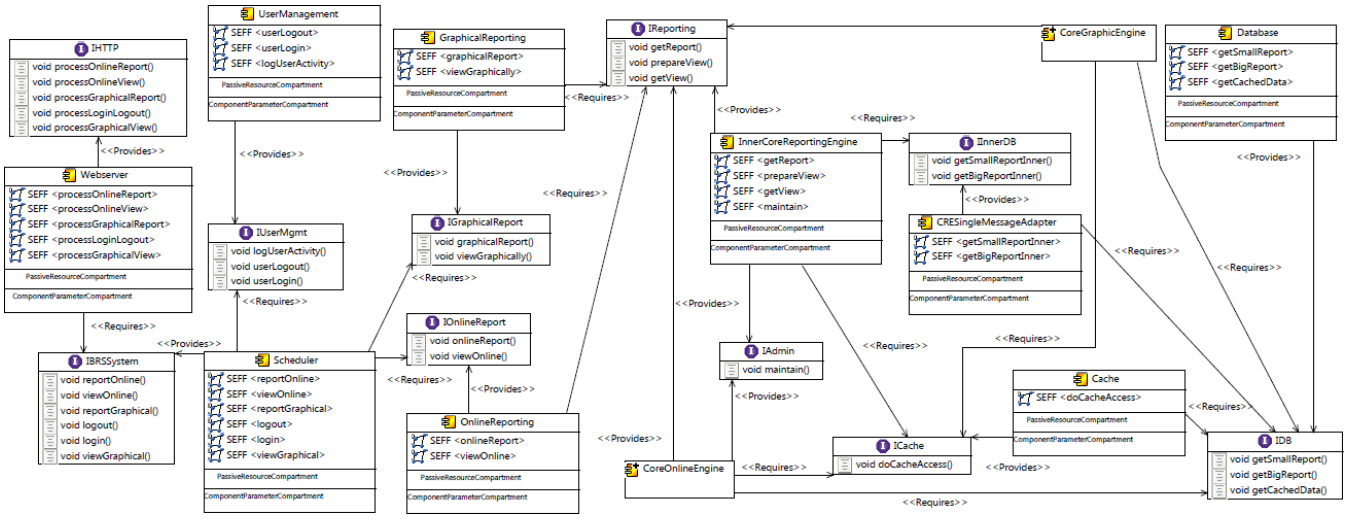


Figure 19: PCM repository resulting from the transformation of the BRS system with UCM2PCM

the models (e.g., converting guards on branches into probabilities or calculating resource demands by substituting the required parameters), since they are not supported by UCM2PCM. The resulting UCMs contained branch probabilities, loop counts, and resource demands.

Because of the missing parameterization concepts in UCM, we used the mean resource demands instead of parameterized resource demands in the model. As discussed in Section 3.9, a mapping of all PCM concepts is neither possible nor intended. Thus, we expect a certain deviation of prediction results.

For the experiments, we transformed the UCMs into PCM models and then ran the discrete-event simulator SimuCom [2]. We also ran SimuCom on the former PCM models and then compared both results. We used Eclipse 3.7, PCM 3.3 running on a Windows 7 PC with 16 GB RAM and an Intel Core i7-2720QM CPU at 2.20 GHz. Each simulation run took less than five minutes to execute.

4.3 Validation of Accuracy

Tab. 1 shows different performance metrics for the systems, both from simulating the original model and from simulating the UCM-based models. The table reports mean and standard deviations of response times and throughputs in seconds, the mean resource utilizations as percentages.

The table shows that the difference between the original model and the UCM-based models is below 10 percent in most cases. The standard deviation of the MediaStore response time shows a difference of 14 percent compared to the original model, which probably could have prevented by using the distribution functions instead of mean values as resource demands. The mean response time for the BRS show an 11 percent higher difference. In this case, the difference resulted from several simplications that had to be introduced into the BRS UCM.

Despite some differences in the simulation results, we deem the accuracy of UCM2PCM sufficient to support early design-time performance decisions. The low differences for different kinds of models demonstrate that UCM2PCM was successful in bridging most semantic differences in these models.

4.4 Validation of Usability

Tab. 2 shows the results from a user survey to evaluate the usability of UCM2PCM. We let each participant (i.e., computer science graduate students and PhD students) work through a tutorial and then model the Media Store system as an exercise. Afterwards, the participants used a web tool to anonymously submit their ratings for the approach and the tool on the following scale: (−−, −, −, +, ++, +++).

While all of the participants were able to quickly produce a UCM, a majority was afraid that UCMs would not be suitable for complex models. This might be a pointer for future work, to further simplify and support the creation of large models with the tool. The comprehensibility of the models was deemed good or very good by all participants, especially for non-experts. Thus is a first hint for the good applicability of the tool for discussing designs with stakeholders from

	Original model	UCM model	% Difference
MediaStore			
Mean ResponseTime in sec (Scenario 1)	2,368	2,366	-0,1%
StdDev Response Time in sec (Scenario 1)	1,705	1,496	-14,0%
Throughput per sec (Scenario 1)	1,267	1,268	0,1%
Utilization (App Server CPU)	58,7%	58,8%	0,2%
Utilization (DB Server CPU)	2,0%	2,0%	0,0%
Utilization (DB Server HD)	87,6%	87,6%	0,0%
SPECjAppServer			
Mean ResponseTime in sec (Scenario 1)	0,026	0,026	-1,0%
StdDev Response Time in sec (Scenario 1)	0,012	0,012	2,2%
Throughput per sec (Scenario 1)	10,407	10,092	-3,1%
Mean Response Time in sec (Scenario 2)	1,044	1,050	0,5%
StdDev Response Time in sec (Scenario 2)	0,014	0,013	-1,9%
Throughput per sec (Scenario 2)	9,626	9,817	1,9%
Utilization (WL Server)	34,7%	34,8%	0,3%
Utilization (Oracle Server)	17,5%	17,6%	0,6%
Business Reporting System			
Mean ResponseTime in sec (Scenario 1)	8,853	7,974	-11,0%
StdDev Response Time in sec (Scenario 1)	3,295	3,163	-4,2%
Throughput per sec (Scenario 1)	0,531	0,540	1,7%
Utilization (Server1)	59,4%	60,2%	1,3%
Utilization (Server2)	65,3%	59,5%	-9,7%
Utilization (Server3)	59,4%	60,4%	1,7%

Table 1: Evaluation results showing simulation results from the original PCM models, and the UCM-based PCM models created by UCM2PCM.

Questions	Participant 1	Participant 2	Participant 3	Participant 4	Participant 5	Participant 6	Participant 7
How long (in minutes) did you evaluate UCM2PCM ?	30	45	40	40	150	30	45
How long (in minutes) did it take you to model the "MediaStore with Cache" as UCM?	45	60	15	15	40	45	120
Speed: How fast basic performance models can be created with the UCM2PCM tool?	++	++	++	++	+	+++	-
Complexity: Is the UCM2PCM tool also suited for creating more complex models?	--	--	++	+	--	-	---
Comprehensibility:Is a UCM model, created with UCM2PCM clear and comprehensible?	++	+	+++	++	+	++	++
Simplification: Is it easier to create performance models using the UCM2PCM tool, compared to modeling with PCM?	-	-	++	++	+	++	+
Speed up: Does the usage of the UCM2PCM tool speeds up performance modeling, compared to modeling with PCM?	+	-	++	++	+	+++	--
Comprehensibility for experts: Are UCMs created with the UCM2PCM tool more comprehensive than PCM models?	++	--	++	-	+	+	+
Comprehensibility for non-experts: Are UCMs created with the UCM2PCM tool more comprehensive than PCM models?	+++	++	+++	++	++	+++	++

Table 2: User Ratings for UCM2PCM

the industrial automation domain.

The results of this user survey are not statistically significant due to the small sample size. Furthermore, the participants might have been biased towards the novel approach. A future empirical study should investigate a larger sample size, analyze different design alternatives, and compare UCM modeling with other methods.

5. RELATED WORK

Surveys on model-based performance prediction [1], performance evaluation of component-based systems [12] and the future of software performance engineering [29] provide a broad overview of recent approaches for performance modeling. In addition, Smith and Williams [25] described fundamentals of software performance engineering and Menasce et al. [16] detailed on capacity planning with queuing models.

Several other approaches focused on model transformations from performance-annotated UML models to various performance models, such as queuing networks [9], stochastic Petri nets [3, 10], and stochastic process algebra [26]. Petriu and Woodside use the UML Schedulability, Performance, and Time (SPT) profile to enrich UML diagrams with performance annotations [19]. In [28] they show the usage of the UML MARTE (Modeling and Analysis of Real-Time and Embedded Systems) profile, which is the successor of the SPT profile.

As UML is the de-facto standard modeling language in the software industry, these approaches have a higher probability of being broadly adopted in practice. However, the UML specification and the UML MARTE specification for performance annotations are complex and therefore difficult to learn for stakeholders from industrial automation. In contrast, UCMs provide a reduced set of model attributes and intuitive visualizations and are thus more applicable in our context. UML mainly focuses on object-oriented software development, while we target component-based development.

KLAPER [7] is a meta model language for performance prediction of component based systems. With the KLAPER Suite, there is a set of tools to actually create performance prediction from the model. This is comparable to the PCM tool chain, but KLAPER is no language to design component-based models. It does not offer a front-end for software design.

UCMs are more abstract than for example UML sequence diagrams. They do not include all message and signals exchanged between components, but can focus on the important, performance-relevant behaviors. Abstracting from unnecessary details is an essential part of modeling and allows to use the notation during early development stages. In contrast to UML activity diagrams, the UCM mapping of responsibilities to software components is arguably more intuitive and visually more appealing than using the UML swimlane notation.

Petriu and Woodside proposed a specific approach for performance modeling with UCMs [21, 20]. They introduced the scenario to performance (S2P) algorithm to transform UCMs into LQNs. S2P allows to transform modeled asynchronous, synchronous, and forwarding requests between software processes to represent common communication patterns in distributed systems. Compared to our approach S2P resulted in a monolithic performance model (LQN), which does not support a component-based development process. It complicates exchanging individual components in the system because of the lacking interface concept. Our target model PCM supports reusable component models as well as a decoupled resource model. The latter allows quickly attaching different resource environment models to the component model so that sizing and capacity planning questions can be answered quickly. Furthermore, tool support for S2P is no longer available.

6. CONCLUSION & FUTURE WORK

Performance engineering is an important part of improving the quality of service of software systems, but usually requires performance specialists to be involved in the development process. The approach presented in this paper lowers the entrance barrier for performance engineering by combining the intuitively understandable modeling language (UCM) with a sophisticated performance engineering approach (PCM). Software engineers can create UCMs and use the defaults in the presented approach to transform them into PCM instances. The simulation and analysis tools of the PCM workbench deliver performance values that can be used for the evaluation of design alternatives.

If more detailed performance properties are required, the generated PCM instances can be further adapted. Thus, the approach enables software engineers to model performance properties on different levels of abstraction. Data-flow oriented systems can be described more easily and clearly than with PCM alone, without losing the benefits of the PCM simulation and analysis techniques.

The integration of UCMs into the PCM development process and modeling workflow has to be investigated further. A reverse transformation that allows developers to (re)transform PCM models into the more comprehensible UCM is desirable. Furthermore, a validation of UCM models is currently lacking at editing time. Once model transformations are fast enough to check the consistency of input models on-the-fly in a similar way to text-based code assistance in state-of-the-art CASE tools, the approach could also be extended to component models other than the PCM, with UCMs serving as an easily understandable modeling language in early phases of the software design process.

7. ACKNOWLEDGMENTS

The authors would like to thank all participants of the user study for their support.

8. REFERENCES

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

[2] S. Becker, H. Koziolek, and R. Reussner. The Palladio component model for model-driven performance prediction. *JSS*, 82:3–22, 2009.

[3] S. Bernardi and J. Merseguer. Performance evaluation of uml design with stochastic well-formed nets. *Journal of Systems and Software*, 80(11):1843–1865, 2007.

[4] F. Brosig, N. Huber, and S. Kounev. Automated Extraction of Architecture-Level Performance Models of Distributed Component-Based Systems. In *26th IEEE/ACM Intl. Conference On Automated Software Engineering (ASE)*, November 2011.

[5] R. Buhr. Use case maps: A new model to bridge the gap between requirements and design. In *OOPSLA workshop – Requirements Engineering: Use Cases and More, Sunday October 15*, 1995.

[6] Chair Software Design and Quality, Karlsruhe Institute of Technology. Palladio: The software architecture simulator. http://www.palladio-simulator.com/, August 2012.

[7] A. Ciancone, A. Filieri, M. Drago, R. Mirandola, and V. Grassi. Klapersuite: An integrated model-driven environment for reliability and performance analysis of component-based systems. *Objects, Models, Components, Patterns*, pages 99–114, 2011.

[8] T. de Gooijer, A. Jansen, H. Koziolek, and A. Koziolek. An industrial case study of performance and cost design space exploration. In *Proc. 3rd Int. Conf. on Performance Engineering (ICPE'12)*, pages 205–216. ACM, April 2012.

[9] A. Di Marco and P. Inverardi. Compositional generation of software architecture performance qn models. In *Proc. 4th Working IEEE/IFIP Conference on Software Architecture (WICSA'04)*, pages 37–46. IEEE, 2004.

[10] S. Distefano, M. Scarpa, and A. Puliafito. From uml to petri nets: The pcm-based methodology. *IEEE Transactions on Softw*, 37(1):65 –79, jan.-feb. 2011.

[11] International Telecommunication Union (ITU). *User requirements notation (URN) - Language definition*, z.151 edition, 11 2008.

[12] H. Koziolek. Performance evaluation of component-based software systems: A survey. *Performance Evaluation*, 67(8):634–658, 2010.

[13] H. Koziolek and R. Reussner. A model transformation from the palladio component model to layered queueing networks. *Performance Evaluation: Metrics, Models and Benchmarks*, pages 58–78, 2008.

[14] H. Koziolek, B. Schlich, C. Bilich, R. Weiss, S. Becker, K. Krogmann, M. Trifu, R. Mirandola, and A. Martens. An industrial case study on quality impact prediction for evolving service-oriented software. In *Proc. 33rd ACM/IEEE Int. Conf. on Software Engineering (ICSE'11) Software Engineering in Practice Track*, pages 776–785. ACM, May 2011.

[15] P. Meier, S. Kounev, and H. Koziolek. Automated transformation of component-based software architecture models to queueing petri nets. In *Proc. 19th IEEE/ACM Int. Symposium on Modeling, Analysis and Simulation of Computer and Telecommunication Systems (MASCOTS'11)*, July 2011.

[16] D. Menasce, V. Almeida, L. Dowdy, and L. Dowdy. *Performance by design: computer capacity planning by example*. Prentice Hall, 2004.

[17] Object Management Group (OMG). Meta Object Facility (MOF) 2.0 Query/View/Transformation (QVT). http://www.omg.org/spec/QVT/Current, January 2011.

[18] Object Management Group (OMG). The UML Profile for MARTE: Modeling and Analysis of Real-Time and Embedded Systems. http://www.omg.org/spec/MARTE/, September 2011.

[19] D. Petriu and M. Woodside. A metamodel for generating performance models fron uml designs. *Lecture notes in computer science*, pages 41–53, 2004.

[20] D. Petriu and M. Woodside. Software performance models from system scenarios. *Performance Evaluation*, 61(1):65–89, 2005.

[21] D. B. Petriu. Layered software performance models constructed from use case map specifications. Master's thesis, Carleton University Ottawa, 2001.

[22] Project SEG. jUCMNav Project Website. http://goo.gl/4JL3K, May 2012.

[23] C. Rathfelder, S. Becker, K. Krogmann, and R. Reussner. Workload-aware system monitoring using performance predictions applied to a large-scale e-mail system. In *Proceedings of the Joint 10th Working IEEE/IFIP Conference on Software Architecture (WICSA) & 6th European Conference on Software Architecture (ECSA)*, Helsinki, Finland, 2012.

[24] R. Reussner, S. Becker, E. Burger, J. Happe, M. Hauck, A. Koziolek, H. Koziolek, K. Krogmann, and M. Kuperberg. The Palladio Component Model. Technical report, KIT, Fakultät für Informatik, Karlsruhe, 2011.

[25] C. Smith and L. Williams. *Performance Solutions: a practical guide to creating responsive, scalable software*, volume 34. Addison-Wesley Boston, MA;, 2002.

[26] M. Tribastone and S. Gilmore. Automatic extraction of pepa performance models from uml activity diagrams annotated with the marte profile. In *Proceedings of the 7th International Workshop on Software and Performance*, pages 67–78. ACM, 2008.

[27] C. Vogel. A use case map editor for rapid performance modeling and reasoning. Master's thesis, Karlsruhe Institute of Technology (KIT), 2012.

[28] M. Woodside. From annotated software designs (uml spt/marte) to model formalisms. In *Proceedings of the 7th international conference on Formal methods for performance evaluation*, pages 429–467. Springer-Verlag, 2007.

[29] M. Woodside, G. Franks, and D. Petriu. The future of software performance engineering. In *2007 Future of Software Engineering*, pages 171–187. IEEE Computer Society, 2007.

[30] X. Wu and M. Woodside. Performance Modeling from Software Components. In *Proc. 4th International Workshop on Software and Performance (WOSP'04)*, volume 29, pages 290–301, New York, NY, USA, 2004. ACM Press.