Modeling of Parametric Dependencies for Performance Prediction of Component-based Software Systems at Run-time

Simon Eismann, Jürgen Walter, Jóakim von Kistowski and Samuel Kounev
University of Würzburg
firstname.lastname@uni-wuerzburg.de

***Abstract*—Model-based performance analysis can be leveraged to explore performance properties of software systems. To capture the behavior of varying workload mixes, configurations, and deployments of a software system requires formal modeling of the impact of configuration parameters and user input on the system behavior. Such influences are represented as parametric dependencies in software performance models. Existing modeling approaches focus on modeling parametric dependencies at design-time. This paper identifies run-time specific parametric dependency features, which are not supported by existing work. Therefore, this paper proposes a novel modeling methodology for parametric dependencies and a corresponding graph-based resolution algorithm. This algorithm enables the solution of models containing component instance-level dependencies, variables with multiple descriptions in parallel, and correlations modeled as parametric dependencies. We integrate our work into the Descartes Modeling Language (DML), allowing for accurate and efficient modeling and analysis of parametric dependencies. These performance predictions are valuable for various purposes such as capacity planning, bottleneck analysis, configuration optimization and proactive auto-scaling. Our evaluation analyzes a video store application. The prediction for varying language mixes and video sizes shows a mean error below 5% for utilization and below 10% for response time.**

***Keywords*-Architecture, Component-based systems, Performance, Performance modeling, Parametric dependencies, Run-time, Descartes modeling language**

I. Introduction

Architectural performance models provide a powerful tool enabling performance prediction for modern component-based software systems. These performance predictions can be used for multiple purposes, such as capacity planning [1] and automated resource management [2]. Providing accurate performance predictions using architectural performance models requires the modeling of several system properties. Specifically, it requires the explicit modeling of dependencies between different model parameters, for example the resource demand of a service might depend on the value of its input parameters. Modeling such parametric dependencies expands the range of system settings that can be accurately modeled [3, 4, 5, 6]. Formal modeling of parametric dependencies allows predicting the impact of changing workloads, system reconfigurations, and deployment changes using a single performance model.

At run-time, variables and dependencies can be learned and continuously updated based on available monitoring data [7]. However, parts of the system may be inaccessible to monitoring as instrumentation might negatively impact the overall system performance [8]. At design-time, many details on component interactions remain open, limiting concrete specifications of parametric dependencies to aspects independent of component instantiation. In contrast, a run-time environment can supply sufficient information (e.g., using monitoring) to identify dependencies for specific component instances.

Existing architectural performance modeling formalisms [5, 9, 10, 11, 12] apply design-time assumptions when modeling parametric dependencies. In particular, we identified the following major limitations: (i) Existing parameter dependency models only support the modeling of dependencies per component type, not between specific component instances. However, in a run-time scenario, where dependencies between specific component instances can be obtained from monitoring data, modeling them becomes necessary. (ii) No support for parametrization based on multiple parametric dependencies, which would allow for selecting alternative input parameters based on what is measurable. (iii) Existing solutions [5, 10, 11] cannot incorporate and benefit from correlations if they are not based on previously executed parameters from the call path, such as backwards correlations.

This paper contributes a novel approach to solve parametric dependencies in architectural performance models of component-based software systems. The dependency resolution algorithm transforms its information to a directed graph and resolves this graph to derive a fully parameterized model. Compared to existing work, our approach enables the resolution of the following features: (i) parametric dependencies on a component instance level, (ii) multiple independent dependencies describing one variable and (iii) modeling of correlations as parametric dependencies.

Applying the presented approach, software architects benefit from an improved ability to reflect system behavior within architectural performance models based on a higher flexibility of inputs. This results in more accurate performance predictions valuable for various purposes such as capacity planning [13], bottleneck analysis [14], configuration

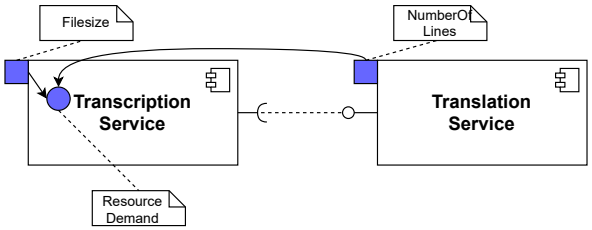

Figure 1: Generation of new subtitles.

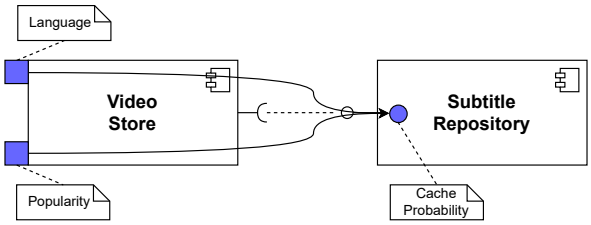

Figure 2: Retrieval of subtitles from the subtitle repository.

optimization [15] and proactive auto-scaling [16].

We implement our methodology and integrate it into the Descartes Modeling Language (DML) [2, 17], an architectural performance model with the goal of enabling run-time resource management.

We evaluate the practical applicability of our approach, based on two case studies in the context of an online video store application. The first case study shows that the CPU resource demand of a component can be described using two independent parameters. The second case study investigates the performance prediction accuracy for different parameter configurations. We show that DML can accurately predict the video store's performance when varying language mixes. Predictions of video store utilization feature a mean absolute percentage error of below 5%, whereas response times can be predicted with an error of below 10%.

II. Motivating example

We illustrate general deficiencies for modeling of parametric dependencies based on a video store application and then derive the general problem statement.

A. Video Store Example

The challenges of modeling parametric dependencies in run-time scenarios can be easily highlighted using an online video store application, similar to YouTube, Netflix or Amazon Video. The video store in our example provides its videos with subtitles in different languages. The subtitles are automatically generated and translated. We introduce two use cases which underline common requirements for modeling parametric dependencies at run-time. The first use case covers the automatic generation of new subtitles by transcribing the audio using machine learning and subsequent translation to different languages. In the second use case, the video store retrieves subtitles from a subtitle repository with a cache.

Use case 1: This use case is inspired by YouTube's 'auto-caps' feature[1]. To automatically generate subtitles for newly uploaded videos, a transcription service uses machine learning techniques to transcribe the video's audio track. Resulting subtitles are then automatically translated by a translation service, as shown in Figure 1. The resource demand of the transcription service can be derived from the size of the file it transcribes. Another way to characterize this resource demand is to reverse engineer it using the resulting number of subtitle lines.

This use case motivates two novel modeling features: First, when designing a performance model it might not be known for which parameters monitoring data will be available at run-time. Therefore, both dependencies have to be modeled and the decision which of these two dependencies should be used to characterize the resource demand of the transcription service should be made at run-time. Secondly, this use-case motivates the modeling of correlations as parametric dependencies. Variables can be described by correlations instead of relying on cause and effect. In general, the output of a method may encapsulate information about its internal execution process. In concrete, the correlation between the number of lines and the transcription resource demand opens an additional valid way to derive the resource demand.

Use case 2: This use case is inspired by video-on-demand providers, such as Netflix or Amazon Video. When a user requests a video, the subtitle repository provides subtitles in the corresponding language, as shown in Figure 2. The subtitle repository implements a cache, which contains the frequently requested subtitles. The retrieval of subtitles from the cache causes a lower resource demand than retrieving subtitles from the database. The cache probability depends on both the popularity and language of the requested video, as subtitles in a frequent language for popular videos are more likely to be in the cache. This interaction can not be generalized for all subtitle repository instances. Other subtitle repository instances might be used in a different context, where the access frequency does not depend on the subtitle language or video popularity. An example for this would be a component creating backups, which iterates over all subtitles, independent of language or popularity. This showcases the need to model dependencies on an instance level in addition to component-level dependencies. At run-time, correlations between parameters can be learned from monitoring data for the deployed system. These correlations are only applicable for specific component instances, not for all instances of a component.

[1] https://googleblog.blogspot.de/2009/11/automatic-captions-in-youtube.html

B. Problem Statement

Illustrated by the video store example, we identified the following general requirements to model parametric dependencies for component-based systems in run-time scenarios:

Instance-level dependencies Instance-level dependencies describe interrelations between component instances. Modeling these dependencies for component types, as supported by existing approaches, would apply the dependency to all instances of the component.

Multiple descriptions The description of parameters using multiple independent parametric dependencies provides alternatives for run-time model parametrization. A typical use case can be, for instance, the specification of component-level dependencies and instance-level dependencies for the same variable.

Correlations as dependencies At a runnable system state, monitoring data may reveal correlations between parameters. Modeling these correlations as dependencies can be used to derive characterizations in case parameters cannot be measured. Existing parameter models can only capture strictly causal correlations as dependencies, since they enforce that a parameter may only depend on prior parameters from the same call path [5, 10, 11].

Existing performance modeling approaches [5, 9, 10, 11, 12] cannot model the parametric dependencies that occur in our video store example. In the following, we propose a modeling and resolution approach that provides native support for the above described modeling features.

III. Parametric Dependency Modeling

We integrate our work for modeling and resolution of parametric dependencies into a representative architectural performance modeling formalism, the Descartes Modeling Language (DML) [2, 3, 17, 18, 19]. Therefore, we briefly introduce its relevant parts. The DML meta-model is made of five sub-models: repository, assembly, resource environment, deployment, and usage profile. The repository model defines blueprints for the software components that get assembled and connected in the assembly model. The deployment model describes how these components are distributed across the hardware resources defined in the resource environment model. The usage profile model contains the workload definition. For a detailed introduction to the DML meta-model we refer to [3].

The integration of novel modeling features requires the identification of the affected meta-model parts. The resource landscape and the deployment do not affect parametric dependencies. Only the application architecture meta-model contains information about parameters and dependencies between them. In the application architecture, components are modeled, instantiated, and connected to other components using interface providing and requiring roles. Every

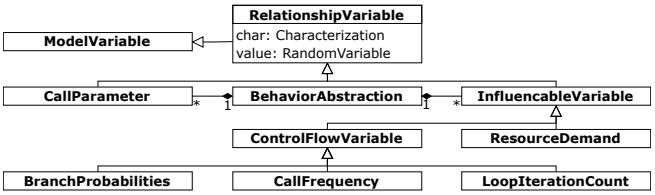

Figure 3: Model variables in DML.

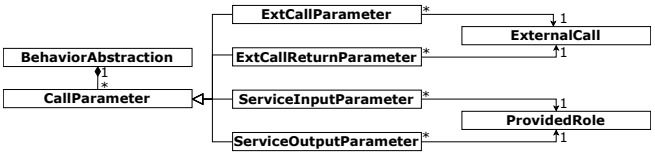

Figure 4: Call parameters in DML.

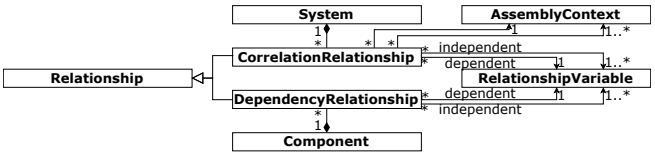

Figure 5: Relationships in DML.

interface signature provided by a component has a corresponding behavior description. It may contain control flow descriptions such as loops, branches, and forks. Additionally, behavior descriptions contain resource demands and external calls to other components. The specification of parametric dependencies builds upon three main concepts: (i) variables, (ii) parameters, and (iii) relationships.

Variables depict a core concept for parametric dependency modeling. Figure 3 shows the variable and parameter types to which dependency descriptions may refer to. Every variable inherits from `RelationshipVariable`. Each `RelationshipVariable` can either be an `InfluencableVariable` or a `CallParameter`, which are both contained in a `BehaviorAbstraction`. An `InfluencableVariable` can be a `ResourceDemand`, a `BranchProbability`, a `CallFrequency`, or a `LoopIterationCount`. Any `RelationshipVariable` contains a `value` attribute of type `RandomVariable` describing its distribution. The `value` attribute can be `NULL` if its distribution is to be derived based on parametric dependencies. The `Characterization` of a `RelationshipVariable` indicates whether the variable value is explicitly modeled or should be characterized using monitoring data.

Calls to components can contain parameters which influence the behavior of the called component. The value returned by a call to another component can also influence the calling component's behavior. Common examples include file sizes and list lengths. In DML, such parameters are modeled as

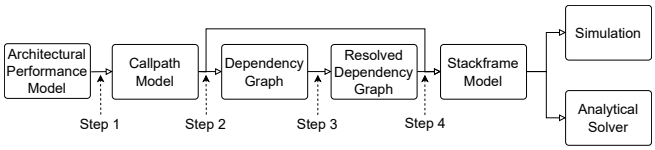

Figure 6: Dependency resolution process.

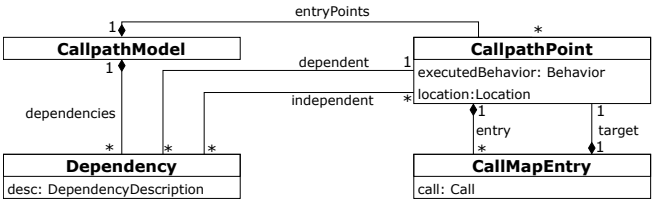

Figure 7: Callpath meta-model.

`CallParameters`, which are depicted in Figure 4. `CallParameters` include `ExtCallParameters`, `ExtCallReturnParameters`, `ServiceInputParameters`, and `ServiceOutputParameters`. `ServiceInputParameters` together with `ServiceOutputParameters` model a component's input and output parameters. Therefore they reference a `ProvidedRole`, which describes the interface and signature the parameter belongs to. The counterparts to these parameters are the `ExtCallParameters` and the `ExtCallReturnParameters`, which specify the input and output parameters for a call to another component. If two component instances are connected via an assembly, the `ExtCallParameter` and the respective `ServiceInputParameter` share the same distribution; the same applies to `ServiceOutputParameters` and `ExtCallReturnParameters`.

In order to model dependencies both on repository level and on component instance level, DML provides two types of `Relationships`, as shown in Figure 5. A `DependencyRelationship` represents a dependency on the repository level and is therefore modeled as part of the component blueprint in the `Repository`. `CorrelationRelationships` on the other hand are used to model dependencies on a component instance level. Therefore it is modeled as part of the `System` and refers to two or more `AssemblyContexts`, which represent specific component instances. Both types of `Relationships` contain an equation which can be used to derive the value of the dependent `RelationshipVariable` from the one or more independent `RelationshipVariables`.

IV. Parametric Dependency Resolution

The second major objective of our approach is to automatically derive performance indices from models using our novel and existing dependency modeling features. The automated prediction requires a resolution of the declarative parametric dependency description to enable model analyses. Besides functional requirements, we formulate the following design goals for the realization of our dependency resolution approach: (i) modularity in order to improve maintainability the resolution and (ii) independence of concrete stochastic model solvers.

To achieve modular design, we decompose the resolution into the multi-step dependency resolution process depicted in Figure 6. It consists of the following steps:

Step 1 Extraction of possible call paths through the system from the architectural model into a `CallpathModel`.

Step 2 Transformation of the model parameters and the dependencies between them into a directed graph, denoted as `DependencyGraph`.

Step 3 Resolution of parametric dependencies within the `DependencyGraph` using our dependency resolution algorithm to generate the `ResolvedDependencyGraph`.

Step 4 Combination of the information contained in the `CallpathModel` and the `ResolvedDependencyGraph` in order to generate a `StackframeModel` [2]. The `StackframeModel` represents a solution-ready model that can be solved using existing simulators and analytical solvers using a transformation to their respective analysis format [2].

The resolution of parametric dependencies should not rely on a specific prediction formalism in order to be reused in transformations to different stochastic models and respective analytical or simulation-based solvers. Through the transformation to the `StackframeModel`, our dependency resolution is independent of specific stochastic model solvers. In addition, this enables the reuse of all existing DML solvers. The description of employed intermediate models allows for their reuse in the context of further architectural performance modeling formalisms.

In the following, we detail the employed intermediate meta-models and our dependency resolution algorithm.

A. Intermediate Meta-Models

The dependency resolution algorithm applies a set of intermediate graph meta-models. In this section, we describe the `CallpathModel`, the `DependencyGraph`, and the `StackframeModel`.

Callpath Model The `CallpathModel`, depicted in Figure 7, captures all paths a request may take through the system. It contains a set of `CallpathPoints` representing entry points to the system. Each `CallpathPoint` specifies the `Location` where its `Behavior` is executed. A location does not refer to a physical node in the system, but to an assembled software component that contains the executed behavior. For every `Call` in the `Behavior` of a `CallpathPoint`, a `CallMapEntry` references another `CallpathPoint` representing the `Call's` target `Behavior` and the assembly component it is executed

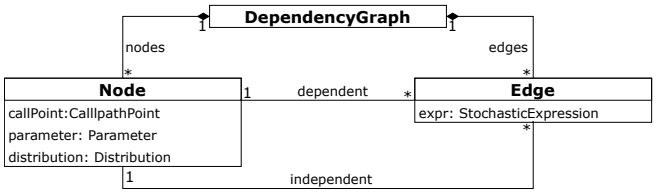

Figure 8: Dependency graph meta-model.

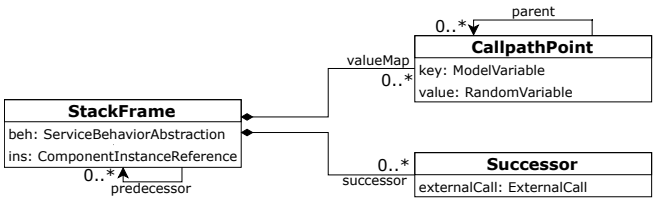

Figure 9: Stackframe model from [2].

in. Every possible path that leads to the execution of a `Behavior` can be described by a single `CallpathPoint` as it contains a reference to its predecessor. Additionally, the `CallpathModel` contains `Dependencies`, which connect two or more `CallpathPoints`. The `Behavior` description of `CallpathPoints` contains the independent and dependent parameters. The nature of the dependency is abstracted in a `DependencyCharacterization`. The meta-model only contains the connections between `CallpathPoints`. Component level dependencies do not have to be modeled in the `CallpathModel` since the location of its dependent and independent components is captured within the model.

Dependency Graph The `DependencyGraph`, depicted in Figure 8, contains information about the model parameters and the dependencies between them. Every `Node` in the Graph represents a `Parameter` and a call path to it, which is represented by a `CallpathPoint`. This means that one `Parameter` can be contained in multiple `Nodes` if it lies on multiple call paths, allowing a `Parameter` to have different values depending on the call path. If the distribution of the `Parameter` in this call path is known the value is saved in its `Distribution`, otherwise it is `NULL`. The `Nodes` are connected by directed `Edges` which model the dependencies between them. Every `Edge` connects one dependent `Node` to one or more independent `Nodes`. The `Edge's StochasticExpression` describes how the distribution of the dependent `Node` can be calculated if the values of the independent `Nodes` are known.

Stackframe Model We reuse the `StackframeModel` presented in [2] as output for our dependency resolution algorithm. The `StackframeModel` represents a solution ready form of an architectural performance model. In the `StackframeModel` the sophisticated aspects of architectural performance models are resolved, simplifying the transformation to solution models such as QPNs or LQNs, as described in [2]. Figure 9 shows the meta-model of the `StackframeModel`. It consists of a series of `StackFrames` which describe a `ServiceBehaviorAbstraction` and a `Component-InstanceReference`, which describe the assembly instance on which the behavior is executed. For every `ExternalCall` in the `StackFrame's` behavior a `Successor` annotates which `StackFrame` is called by the `ExternalCall`. Similarly, a `ValueMapEntry` annotates every `ModelVariable` in the `StackFrame's` behavior with a `RandomVariable` describing its distribution.

B. Resolution Process

This section details how our dependency resolution algorithm uses the intermediate meta-models to transform a model containing parametric dependencies to an analysis-ready model without dependencies. The approach consists of the following four steps:

Step 1 The `CallpathModel` can be extracted from an architectural performance model by creating `Callpath-Points` for all possible entry points. Next, iterating over all newly created `CallpathPoints`, a `CallMapEntry` and a target `CallpathPoint` has to be created for every `Call` inside the `CallpathPoint's Behavior`. This needs to be recursively repeated for all `CallpathPoints` created by this step, until every call path ends in a `Callpath-Point` who's `Behavior` does not contain any `Calls`.

Step 2 The `CallpathModel` can be transformed to the `DependencyGraph` by iterating over all `Callpath-Points` in the `CallpathModel` and executing the following tasks on them:

- Create a `Node` for every parameter in the `Callpath-Point Behavior`.
- Create an `Edge` for every component level dependency described in the `CallpathPoint's Behavior`.
- Create an `Edge` for every pair of input and output parameter in the `CallpathPoint's Behavior` and its predecessor's `Behavior`.
- Create an `Edge` for every instance level dependency modeled in the `CallpathModel` who's dependent is contained in the `CallpathPoint's Behavior`.

This generates a complete `DependencyGraph`, which can be used to derive values for all unknown parameters in the following step.

Step 3 The `DependencyGraph` contains `Nodes` with known distributions and `Nodes` with unknown distributions. If all independent `Nodes` of an `Edge` have distributions the distribution of the dependent `Node` can be derived. To determine a resolution order which resolves distributions for all `Nodes` we use the dependency solver algorithm shown in Algorithm 1. It iterates over all `Edges` in the dependency graph and evaluates if every independent `Node` of the `Edge` is already characterized. Should this be the case

Algorithm 1 Dependency Resolution Algorithm

```
1: function RESOLVEDEPENDENCIES(dependencyGraph)
2:     hasChanged = true
3:     while hasChanged == true do
4:         hasChanged = false
5:         for Edge in dependencyGraph.edges do
6:             allIndependentsCharacterized = true
7:             for Node in Edge.independents do
8:                 if Node.value == null then
9:                     resolvable = false
10:                end if
11:            end for
12:            if resolvable == true then
13:                Edge.dependent.value = Edge.calc()
14:                hasChanged = true
15:            end if
16:        end for
17:    end while
18: end function
```

the algorithm computes the distribution of the dependent `Node` of the `Edge`. The algorithm repeats these steps until no changes occur in the dependency graph after iterating over the `Edges`. This resolves distributions for all `Nodes` and therefore for every occurrence of every parameter in the system.

Step 4 We transform the `CallpathModel` and the `ResolvedDependencyGraph` to a `StackframeModel`. The `CallpathPoints` and the `CallMapEntries` connecting them can be directly mapped to `StackFrames` and `Successor`. For every parameter in a `CallpathPoint Behavior`, its distribution can be found in the corresponding `Node` of the `ResolvedDependencyGraph`. We transform it to a `ValueMapEntry` of the `StackframeModel`. The result of the whole process is a fully parameterized `StackframeModel`, which can be analyzed using a variety of analytical approaches and simulations [2].

V. Evaluation

The evaluation goal is to show the suitability of the proposed run-time dependency modeling features and the capability of our approach to model and solve these parametric dependencies correctly. We refine our evaluation goal using the following research questions, to be evaluated on the previously introduced use cases:

RQ 1 *Can model variables be accurately described by correlations when causal parameters cannot be measured?*

RQ 2 *Can we provide multiple characterizations of a parametric resource demand based on alternative to parameters that enable accurate performance predictions?*

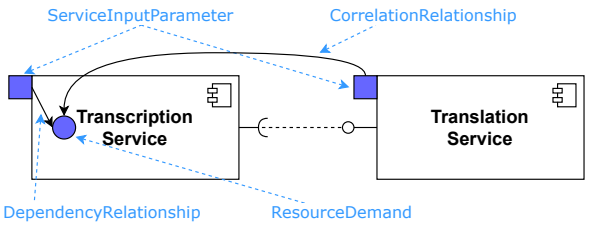

Figure 10: Modeling the video transcription case study.

RQ 3 *Does modeling parametric dependencies on component instance level improve the prediction accuracy?*

To evaluate the research questions, we implemented parts of the motivating video store required for analysis. All of our experiments run on an HPE ProLiant DL160 server. It features an Intel Xeon E5-2650 v3 processor with 10 cores at 2.4 GHz and 32 GB RAM. After dependency resolution, we solve the resulting `StackFrame` model reusing a transformation to QPN and subsequent simulation using SimQPN, a discrete event simulation tool for QPNs [20].

A. Case Study 1 - Video Transcription

1) Setting: To provide an implementation of the video transcription service, we integrate the established open source speech recognition software CMUsphinx [2] into a Java EE servlet application. The processing time of the transcription service depends on the properties of the respective video. The length of a video, as well as the number of subtitles (counted as number of lines), impact the processing overhead. While the number of subtitle lines can be monitored from the Java interface, the monitoring of the file size requires a more intrusive monitoring. For the evaluation we collected a training data set and an evaluation data set, each consisting of 25 videos with 10 to 30 seconds of English spoken content from YouTube. Based on these videos, we investigate the response times of the transcription service for a mixture of videos.

2) Modeling: Figure 10 models the video transcription use case. The CPU resource demand of the `TranscriptionService` is modeled as a `ResourceDemand` variable. Two `ServiceInputParameters` specify the file size and the number of lines parameters. Further, we model a `DependencyRelationship` describing the relationship between the file size and the resource demand on component-level. To describe the relationship between the number of lines and the resource demand of the `TranscriptionService`, we select a `CorrelationRelationship` to model the instance-level dependency.

3) Dependency characterization: To characterize the parametric dependencies, we measure the video file size, the amount of the generated subtitles in number of lines,

[2]https://cmusphinx.github.io/

Input	Polynomial order	p-value	Residual std. error	Multiple R-Squared
File size	1	8.5e-13	3023	0.896
	2	1.4e-11	3078	0.897
	3	6.3e-11	3017	0.906
	4	9.6e-11	2826	0.921
Number of lines	1	1.8e-11	3445	0.865
	2	1.7e-11	3363	0.877
	3	5.2e-10	3338	0.884
	4	2.9e-09	3357	0.889

Table I: Evaluation of the CPU resource demand estimations.

Region	ENG	SPA	GER	FR	IT	POL	RUM
USA	83%	17%	0%	0%	0%	0%	0%
EU	25%	11%	21%	15%	14%	9%	5%

Table II: Assumed language mix within the workload.

Popularity Class	Number of Products	Product Likelihood
Frequent access	200	0.3%
Moderate access	800	0.033%
Long tail	9000	0.0011%

Table III: Product popularity classes.

and corresponding resource demands for the training data set. Then, we fit polynomial functions up to fourth order that describe the resource demand from the file size and the number of lines. Our investigations show that increasing polynomial degrees does not significantly improve the prediction. Therefore, we conclude a linear correlation and model both aspects using first order functions. The CPU resource demand description of the transcription service based on the file size in KB is:

$$ResourceDemand = 43.2 * FileSize - 1662.5$$

Additionally, we model the resource demand based on the number of lines parameter as:

$$ResourceDemand = 2253.5 * NumberOfLines + 3894.2$$

In this use case, modeling the correlation between the number of lines parameter and the transcription service resource demand allows to accurately estimate the resource demand. This provides an additional description of the resource demand, which can be used as a fallback in case the size of the video files cannot be monitored.

4) Experiment results: Table I shows the residual standard error, multiple R-squared metric and p-value for estimations of four polynomial orders with either the file size or the number of subtitle lines as input. The low p-value indicates a significant relationship between the input parameter and the response time, which justifies modeling it. The residual standard error of the number of lines estimation exceeds the error of file size estimation by about 10%. Similarly, the file size estimation captures the variance within the sample slightly better, as described by the multiple R-squared metric. While the file size estimation produces more accurate predictions, the estimation using the number of lines is sufficient in case the file size cannot be monitored. Our experiments show that the correlation between the resource demand and the number of lines parameter provides an accurate description of the resource demand (RQ 1). Moreover, we demonstrate that we can provide two accurate description of the same variable based on different input (RQ 2).

To demonstrate that each of the two descriptions allows for accurate performance predictions, we analyze the accuracy of response time predictions for each description. We configure an exponentially distributed request inter-arrival rate with an average delay of 60 s which corresponds to a CPU utilization around 45%. The requests randomly select a video from the evaluation data set. The measured average response time was 44207 ms. The model predicts an average response time of 45518 ms when using the file size estimation and 46252 ms for the number of lines estimation. This corresponds to an accuracy of 97.03% and 95.38% respectively. While using the file size estimation achieves a better accuracy, the accuracy of both predictions is sufficient for capacity planning [13].

B. Case Study 2 - Subtitle Provider

1) Setting: The second case study monitors the impact of different parametrizations on the system performance for the subtitle retrieval. At the retrieval, subtitles have to be queried from a database if not cached within the subtitle repository component. Subtitles in frequent languages and for popular videos are more likely to be in the cache. Consequently, the likelihood of a subtitle being in the cache is influenced by the requested language as well as the popularity of the video. We assume two workload mixes depicted in Table II, denoted as USA and EU. We assume American traffic to be predominated by English and Hispanic customers, whereas the European traffic consists of a wide variety of languages. The popularity of videos follows a long tail distribution [21]. As shown in Table III, the store provides 10.000 videos from which the 200 most frequently accessed videos make up for over half of the traffic. The next 800 most frequently accessed videos cause 30% of the traffic. The remaining 9000 long tail videos cause only 10% of the traffic. For each video, subtitles can be requested in the seven languages. While the cache allows to store 250 subtitles at a time, the number of subtitles sums up to a total of 70.000. Next, we evaluate how caching behavior affects performance indices when moving applications from one region to another.

2) Modeling: Figure 11 shows modeling of the subtitle provider use case. The behavior of cache hits and misses is modeled as a `BranchAction` with an unknown `BranchingProbability`. The branch denoting a cache-hit is connected to a lower resource demand

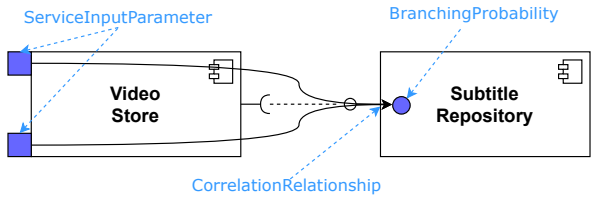

Figure 11: Modeling the subtitle generation case study.

Region	Metric	Eval.	Load-Level high	med	low	lowest
EU	Utilization	measured	79.6	39.2	19.7	9.4
USA	Utilization	measured	57.2	29.9	15.4	8.0
Both	Utilization	predicted	70.7	35.3	17.6	8.8
EU	Relative Prediction Error		11.2	9.9	10.7	5.9
USA			23.6	18.1	14.3	10.6

Table IV: Comparison of the measured utilization to a model-based prediction without parametric dependencies.

than the branch for cache misses. The input parameters `Language` and `Popularity` are modeled as `Service-InputParameters`. The dependency between these parameters and the branching probability is represented by a `CorrelationRelationship` with two independents, to model an instance-level dependency.

3) Dependency characterization: The dependency between the language distribution, the popularity distribution and the probability of accessing the database to retrieve a subtitle follows the Wallenius' noncentral hypergeometric distribution [22], which can be approximated as a binomial distribution. This approximation leads to the following formula describing the probability for a cache hit:

$$\mathrm{P}(hit) = \sum_{l}^{Languages} \sum_{p}^{Popularities} \frac{\mathrm{P}(p)^2 * \mathrm{P}(l)^2 * 250}{|p|} \quad (1)$$

For all popularity levels p and all languages l, their respective occurrence probabilities P(p) and P(l) are squared and multiplied by the cache capacity, which is 250. Finally, the resulting value is divided by the number of subtitles in the language l and popularity class p, which is equivalent to the number of videos in the popularity class p, denoted as $|p|$.

Instance-level dependencies are required to capture the influence of the video popularity and subtitle language influence on the cache probability of the subtitle provider. We argue that this dependency has to be modeled on component instance level, i.e. for this specific component instance. In contrast, the traditional approach would model it as a dependency on repository component level, i.e. a dependency that applies to all subtitle provider instances. However, this dependency can not be applied to every subtitle provider instance. For example, if a subtitle provider instance is used by a backup component the dependency would not apply. The backup component iterates over all subtitles leading to a cache probability of zero, since no subtitle is requested twice. However, the dependency from Section V-B3 would derive a cache probability > 0. This shows that modeling some dependencies on component instance level instead of repository component level is necessary (RQ 3).

Region	Metric	Eval.	Load-Level lowest	low	med	high
EU	Utilization	measured	9.4	19.7	39.2	79.6
		predicted	9.8	19.8	39.0	79.3
	Resp. time	measured	21.0	27.0	39.0	149.0
		predicted	23.1	26.7	37.0	106.0
USA	Utilization	measured	8.0	15.4	29.9	57.2
		predicted	7.2	14.4	28.8	57.4
	Resp. time	measured	17.0	18.0	22.0	45.0
		predicted	16.2	18.3	23.6	43.9

Table V: Comparison of measured performance metrics to predictions using parametric dependencies.

4) Experiment results: Modeling this case study without parametric dependencies corresponds to using the average monitored value for the branching probability. Analysis results presented in Table IV show this to be highly inappropriate to capture the system behavior due to an average utilization prediction error of 13%.

When modeling the dependency between the language and popularity distribution and the branching probability of the subtitle repository, the DML model is capable of accurately predicting the utilization for all eight scenarios, with a relative error below 5%, as shown in Table V. The response time prediction error is below 10%, except for a high load of EU traffic. Here, the model-based prediction underestimates the response time by 28.86%. This outlier is still within a 30% margin considered to be acceptable for capacity planning [13]. By modeling the cache probability as a parametric dependency, our model can accurately predict the impact of different language distributions on the performance of the system.

VI. Limitations and Threats to Validity

While our approach presents an improvement compared to the state of the art, there still are limitations and threats to the validity to be discussed.

Algorithm 1 for the resolution of parametric dependencies always terminates, since each iteration reduces the number of nodes without characterization. In case no new node was characterized, the algorithm also terminates.

Whenever a parametric performance model contains cyclic dependencies, one of the dependencies making up the cycle can be ignored since it describes an already known variable. Assuming that all dependencies describe the variable with the same accuracy, no information is lost.

In case the parameter model contains insufficient information to derive distributions for all variables, our algo-

rithm still resolves as many distributions as possible and then returns the partially resolved `DependencyGraph`. This partially resolved `DependencyGraph` contains the information which variables could not be resolved and measurement values for which parameters would allow to resolve the missing distributions.

The distributions of two variables can be derived from the same parameter or share a common ancestor in the `DependencyGraph`. Then the resulting model wrongly assumes that the distributions of the two variables are statistically independent. This problem is also inherent to existing approaches [23]. In contrast to existing work, our approach allows to automatically detect this by checking whether two variables share a common ancestor in the `DependencyGraph`. This allows to proactively inform the user that the parameterization might be inaccurate.

VII. Related Work

Related work discusses modeling parametric dependencies as well as the identification and characterization of parametric dependencies.

A. Meta-Models

Existing surveys on component-based performance modeling [29, 30] neglect a detailed discussion on capabilities to model parametric dependencies. We categorize existing performance modeling approaches according to their parametric dependency modeling capabilities into four categories:

a) Modeling formalisms having *no parameter model* lack modeling features to model parameters or their impact on the performance of individual components. Such models cannot capture changing workload mixes and system configurations within a single model. Examples include stochastic formalisms, like QN, QPN, LQN and process models, as well as some architectural models [31, 32, 33].

b) Meta-models with a *static parameter model*, as presented in [24, 25, 26, 27], allow for the parametrization of component instances to model the influence of instance-specific parameters on component performance. However, these approaches require each component instance to be parameterized individually. They lack features to model the impact of deployment changes on component performance. Instance-specific parameters can be hardware parameters, such as processing rate or number of cores, or software parameters, such as operating system, hypervisor, or number of virtual cores [26]. Parameters can also be used to model different component configurations, as for example, two instances of a compression component with different compression rates. Resource demands and external call frequencies can be described as a linear combination of the static parameters [25].

c) *Dynamic parameter models* are supported e.g., by PCM [5], RESOLVE [9], HAMLET [10], ROBOCOP [11]. While these models suit for design time analysis, they encounter deficiencies at run-time. This class of models allows the modeling of input [11] and output parameters for components and external calls [5]. The resolution of parameter values propagates from caller to callee which makes parameter values dependent on the deployment context. Correlations that do not propagate from caller to callee cannot be modeled.

d) *Persistent parameter models* allow to model internal state for subsystems and components based on parameters [28]. These parameters may change during the model simulation impacting the component behavior. Modeling the internal state increases the prediction accuracy, but cannot accurately model the case studies presented in this paper.

Table VI compares the capabilities of existing modeling approaches with regard to modeling of parametric dependencies. Summarizing, none of the related approaches provides means to model correlations as dependencies, instance-level dependencies, or multiple descriptions of the same variable.

B. Model Learning

Parametric dependencies can be identified and characterized based on expert knowledge but also automatically based on static code analysis, correlation analysis, or machine learning. Automation reduces the overhead and required expertise to benefit from parametric dependencies. The following approaches can be transferred to the novel modeling features proposed in this work. Krogmann et al. [6] propose to combines static code analysis, micro benchmarks and a genetic search to automatically identify and characterize parametric dependencies. If static code analysis is not feasible, [19] proposes a technique to automatically characterize known dependencies based on monitoring data. Identifying the parameters influencing a target variable, is an established research topic in the area of machine learning called feature selection. Chandrashekar et al. [34] present a comprehensive survey on the state-of-the-art for feature selection.

To minimize the modeling effort for large scale systems, we propose to combine these approaches with fully automated model extraction approaches [35, 36] that are capable to derive non-parametric models.

VIII. Conclusion

This work provides novel modeling features for parametric dependencies and a corresponding solution algorithm, which extends state-of-the-art modeling with run-time specific features. In particular, it enables parametric dependencies on component instance level, multiple descriptions of a single variable, and modeling of correlations as dependencies. Applying the presented approach, software architects benefit from an improved ability to reflect system behavior within architectural performance models based on a higher flexibility of inputs. This results in more accurate performance predictions valuable for various purposes such as capacity planing or proactive auto-scaling. We evaluated

Modeling Feature	No Parameter Model [24, 25, 26, 27]	Static Parameter Model [24, 25, 26, 27]	Dynamic Parameter Model [5, 9, 10, 11, 12]	Persistent Parameter Model [28]	Modeling Approach of this Paper
Input and output parameters	×	×	✓	✓	✓
Component-level dependencies	×	×	✓	✓	✓
Dependency chaining	×	×	✓	✓	✓
Instance-level dependencies	×	×	×	×	✓
Multiple descriptions	×	×	×	×	✓
Correlations as dependencies	×	×	(✓)	(✓)	✓

Table VI: Support of parametric dependency modeling features by existing performance modeling approaches.

our methodology based on two case studies. The first case study shows that multiple dependencies based on different input parameters can be used to describe the same variable. The second case study models parametric dependencies on a component instance level to characterize and accurately predict system behavior. Across both case studies, the mean prediction error for utilization and response time is mostly below 5% and 10% respectively.

IX. Acknowledgements

This work was funded by the German Research Foundation (DFG) under grant No. (KO 3445/11-1). The authors would like to thank the anonymous reviewers for their valuable comments and suggestions.

References

[1] S. Becker, H. Koziolek, and R. Reussner, "The palladio component model for model-driven performance prediction," *Journal of Systems and Software*, vol. 82, no. 1, pp. 3–22, 1 2009.

[2] N. Huber, F. Brosig, S. Spinner, S. Kounev, and M. Bähr, "Model-based self-aware performance and resource management using the descartes modeling language," *IEEE Transactions on Software Engineering*, vol. 43, no. 5, pp. 432–452, 2017.

[3] F. Brosig, N. Huber, and S. Kounev, "Architecture-Level Software Performance Abstractions for Online Performance Prediction," *Elsevier Science of Computer Programming Journal (SciCo)*, vol. 90 Part B, pp. 71–92, 2014.

[4] E. Bondarev, M. Chaudron, and E. de Kock, "Exploring performance trade-offs of a jpeg decoder using the deepcompass framework," in *Proceedings of the 6th International Workshop on Software and Performance*, 2007, pp. 153–163.

[5] H. Koziolek, "Parameter dependencies for reusable performance specifications of software components," Ph.D. dissertation, Universität Oldenburg, 2008.

[6] K. Krogmann, M. Kuperberg, and R. Reussner, "Using genetic search for reverse engineering of parametric behavior models for performance prediction," *IEEE Transactions on Software Engineering*, vol. 36, no. 6, pp. 865–877, 2010.

[7] S. Spinner, "Self-aware resource management in virtualized data centers," Ph.D. dissertation, University of Würzburg, Germany, 2017.

[8] J. Ehlers, A. van Hoorn, J. Waller, and W. Hasselbring, "Self-adaptive software system monitoring for performance anomaly localization," in *Proceedings of the 8th ACM International Conference on Autonomic Computing*, 2011, pp. 197–200.

[9] M. Sitaraman, G. Kulczycki, J. Krone, W. F. Ogden, and A. L. N. Reddy, "Performance specification of software components," *SIGSOFT Softw. Eng. Notes*, vol. 26, no. 3, pp. 3–10, 2001.

[10] D. Hamlet, "Tools and experiments supporting a testing-based theory of component composition," *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol. 18, no. 3, p. 12, 2009.

[11] E. Bondarev, P. de With, M. Chaudron, and J. Muskens, "Modelling of input-parameter dependency for performance predictions of component-based embedded systems," in *31st EUROMICRO Conference on Software Engineering and Advanced Applications*, 2005, pp. 36–43.

[12] P.-O. Ostberg, H. Groenda, S. Wesner, J. Byrne, D. S. Nikolopoulos, C. Sheridan, J. Krzywda, A. Ali-Eldin, J. Tordsson, E. Elmroth *et al.*, "The cactos vision of context-aware cloud topology optimization and simulation," in *Cloud Computing Technology and Science (CloudCom)*, 2014, pp. 26–31.

[13] D. A. Menasce and A. F. A. Virgilio, *Scaling for E Business: Technologies, Models, Performance, and Capacity Planning*, 1st ed. PTR, 2000.

[14] N. Sato and K. S. Trivedi, "Stochastic modeling of composite web services for closed-form analysis of their performance and reliability bottlenecks," in *International Conference on Service-Oriented Computing*, 2007, pp. 107–118.

[15] A. Koziolek, D. Ardagna, and R. Mirandola, "Hybrid multi-attribute QoS optimization in component based software systems," *Journal of Systems and Software*, vol. 86, no. 10, pp. 2542 – 2558, 2013.

[16] A. Bauer, N. Herbst, and S. Kounev, "Design and Evaluation of a Proactive, Application-Aware Auto-Scaler," in *Proceedings of the 8th ACM/SPEC International Conference on Performance Engineering (ICPE 2017)*, 2017, pp. 425–428.

[17] S. Kounev, N. Huber, F. Brosig, and X. Zhu, "A Model-Based Approach to Designing Self-Aware IT Systems and Infrastructures," *IEEE Computer*, vol. 49, no. 7, pp. 53–61, 2016.

[18] N. Huber, A. van Hoorn, A. Koziolek, F. Brosig, and S. Kounev, "Modeling run-time adaptation at the system architecture level in dynamic service-oriented environments," *Service Oriented Computing and Applications*, vol. 8, no. 1, pp. 73–89, 2014.

[19] F. Brosig, N. Huber, and S. Kounev, "Automated extraction of architecture-level performance models of distributed component-based systems," in *Proceedings of the 2011 26th IEEE/ACM International Conference on Automated Software Engineering*, 2011, pp. 183–192.

[20] S. Kounev and A. Buchmann, "SimQPN - a tool and methodology for analyzing queueing Petri net models by means of simulation," *Performance Evaluation*, vol. 63, no. 4-5, pp. 364–394, 2006.

[21] E. Brynjolfsson, Y. J. Hu, and M. D. Smith, "From niches to riches: The anatomy of the long tail," *Sloan Management Review*, vol. 47, no. 2, pp. 67–71, 2006.

[22] A. Fog, "Calculation methods for wallenius' noncentral hypergeometric distribution," *Communications in Statistics-Simulation and Computation®*, vol. 37, no. 2, pp. 258–273, 2008.

[23] F. Brosig, P. Meier, S. Becker, A. Koziolek, H. Koziolek, and S. Kounev, "Quantitative Evaluation of Model-Driven Performance Analysis and Simulation of Component-based Architectures," *IEEE Transactions on Software Engineering (TSE)*, vol. 41, no. 2, pp. 157–175, 2015.

[24] X. Wu and M. Woodside, "Performance modeling from software components," in *Proceedings of the 4th International Workshop on Software and Performance*, 2004, pp. 290–301.

[25] H. Gomaa and D. A. Menascé, *Performance Engineering of Component-Based Distributed Software Systems*. Springer Berlin Heidelberg, 2001, pp. 40–55.

[26] S. Zschaler, "Formal specification of non-functional properties of component-based software systems," *Software and Systems Modeling*, vol. 9, no. 2, pp. 161–201, 2010.

[27] Y. Liu, I. Gorton, and A. Fekete, "Design-level performance prediction of component-based applications," *IEEE Transactions on Software Engineering*, vol. 31, no. 11, pp. 928–941, 2005.

[28] L. Happe, B. Buhnova, and R. Reussner, "Stateful component-based performance models," *Softw. Syst. Model.*, vol. 13, no. 4, pp. 1319–1343, 2014.

[29] S. Balsamo, A. Di Marco, P. Inverardi, and M. Simeoni, "Model-based performance prediction in software development: A survey," *IEEE Transactions on Software Engineering*, vol. 30, no. 5, pp. 295–310, 2004.

[30] H. Koziolek, "Performance evaluation of component-based software systems: A survey," *Performance Evaluation*, vol. 67, no. 8, pp. 634–658, 2010.

[31] D. Garlan, R. Monroe, and D. Wile, "Acme: An architecture description interchange language," in *Proceedings of the 1997 Conference of the Centre for Advanced Studies on Collaborative Research*, 1997, pp. 7–14.

[32] A. Mos and J. Murphy, "A framework for performance monitoring, modelling and prediction of component oriented distributed systems," in *Proceedings of the 3rd International Workshop on Software and Performance*, 2002, pp. 235–236.

[33] K. Wallnau and J. Ivers, "Snapshot of ccl: A language for predictable assembly," Software Engineering Institute, Carnegie Mellon University, Pittsburgh, PA, Tech. Rep., 2003.

[34] G. Chandrashekar and F. Sahin, "A survey on feature selection methods," *Computers & Electrical Engineering*, vol. 40, no. 1, pp. 16–28, 2014.

[35] A. Brunnert and H. Krcmar, "Continuous performance evaluation and capacity planning using resource profiles for enterprise applications," *Journal of Systems and Software*, vol. 123, pp. 239–262, 2017. [Online]. Available: http://dx.doi.org/10.1016/j.jss.2015.08.030

[36] J. Walter, C. Stier, H. Koziolek, and S. Kounev, "An Expandable Extraction Framework for Architectural Performance Models," in *Proceedings of the 3rd International Workshop on Quality-Aware DevOps (QUDOS'17)*. ACM, 4 2017.