Julius-Maximilians-
UNIVERSITÄT
WÜRZBURG

Solving Explicit Dependencies for Architectural Performance Models

Master Thesis of

Simon Eismann

Department of Computer Science
Chair for Computer Science II
Software Engineering

Reviewer: Samuel Kounev
Advisor: Dipl Inform Jürgen Walter

Duration: 14. December 2016 – 14. June 2017

Contents

1 Introduction **5**

2 Foundations **9**
2.1 Parametric Dependencies 9
2.2 Descartes Modeling Language (DML) 10
2.3 Parametric Dependencies in DML 15
2.4 Queuing Petri Net (QPN) 16
2.5 SimQPN 16
2.6 DML Solution Process 17

3 Related Work **21**
3.1 Modeling of Parametric Dependencies 21
3.2 Extraction of Parametric Dependencies 22
3.3 Resolution and Calculation of Parametric Dependencies 23

4 Approach **25**
4.1 Goals 25
4.2 Approach 27

5 Solving Parametric Dependencies **29**
5.1 Callpath Model 29
5.2 Dependency Graph 30
5.3 Relationship Graph Solving 31
5.4 Dependency Calculation 31
Probabilistic Logic Values 32

6 Application of Parametric Dependency Solving to DML **39**
6.1 Callpath Model 39
6.2 Callpath Model Extraction 40
6.3 Relationship Graph 41
6.4 Relationship Graph Extraction 42
6.5 Dependency Calculation 42

7 Implementation **47**
7.1 Components 47
7.2 Control flow 48
7.3 Integration in existing solver 48
7.4 Testing 50
7.5 Technical Details 50

8 Evaluation **53**
8.1 Goal Question Metric (GQM) 53
8.2 Methodology 54

8.3 Dependency Calculation Evaluation 55
8.4 Dependency Resolution Evaluation 57

9 Conclusion **65**
9.1 Summary 65
9.2 Benefit 66
9.3 Future Work 67

10 Acronyms **75**

Bibliography **77**

Abstract

Architectural performance models can be leveraged to explore performance properties of software systems. While non-parametric component behavior can be easily modeled and predicted, parametric behavior causes additional complexity for performance modeling not supported by all Software Performance Engineering (SPE) approaches. The Palladio Component Model (PCM) requires to model parameters and parametric dependencies explicitely which might be only partly known to performance engineers especially in online scenarios. The parameter model of Descartes Modeling Language (DML) allows to model scenarios in which not all parameters are known, but did not yet provide a solver. This master thesis discusses resolution of parametric dependencies in general and for DML in particular. The resolution consists of the extraction of a directed graph containing all parameters and dependencies including the application of arithmetic operations on distributions. Our implementation derives performance predictions by solving the resulting graph based on a flooding algorithm, a transformation to QPN, and subsequent simulation. We provide an end-to-end evaluation of our approach using a media store model. Moreover, we evaluate modeling alternatives using benchmark models and comparing merged distributions to empirically created samples.

Zusammenfassung

Architekturelle Performance Modelle können genutzt werden um die Performance Eigenschaften eines Software Systems zu untersuchen. Während nicht parametrisches Verhalten von Komponenten einfach modelliert und vorhergesagt werden kann, verursacht parametrisches Verhalten zusätzliche Komplexität bei der Modellierung die nicht von allen Software Performance Engineering (SPE) Ansätzen unterstützt wird. Das Palladio Component Model (PCM) benötigt eine explizite Modellierung der Parameter und der parametrischen Abhängigkeiten. In einem Online Szenario ist es wahrscheinlich, dass nicht all diese Informationen dem Performance Engineer zur Verfügung stehen. Das Parameter Modell der Descartes Modeling Language (DML) erlaubt die Modellierung mit unvollständigen Informationen, aber stellt keinen Lösungsansatz dafür zur Verfügung. Diese Master Arbeit diskutiert generell die Auflösung von parametrischen Abhängigkeiten und die Anwendung dieser Ideen für DML spezifisch. Die Auflösung besteht aus der Extraktion eines gerichteten Graphen der all Parameter und Abhängigkeiten zwischen ihnen enthält. Unsere Implementierung generiert Performance Vorhersagen durch das Auflösen des Graphen mit Hilfe eines Flooding Algorithmus und einer darauffolgenden Transformation zu Warteschlangen-Petri-Netzen und Simulation. Wir evaluieren unseren Ansatz von Ende zu Ende mithilfe eines Media Store Modells. Des weiteren evaluieren wir verschiedene Modellierungsfeatures mithilfe von Benchmark Modellen und wir vergleichen die berechneten Verteilungen mit empirisch erzeugten Samples.

1. Introduction

Modern software systems allow for dynamic reconfiguration at runtime [AFG+10]. Technologies like containerization enable flexibly replication of component instances. Virtualization allows to abstract the physical layer into virtual machines on which applications can be deployed. Due to these and other technologies, software systems can be dynamically reconfigured. Due to changing workloads continuous adaptations are required to constantly tune configuration for resource efficiency while ensuring quality of service. Commonly, adaptations are automated using either reactive or pro-active approaches. Reactive approaches define upper and lower thresholds for performance metrics like utilizations and scale the system on violations. Reactive approaches tend to either scale late causing Service Level Agreement (SLA) violations or to overprovision massively. Proactive approaches apply predictions of both the workload and the system's performance to adapt to changes in the workload before they happen in production. Reactive autoscalers are commonly used in industry, but in some scenarios they start to oscillate [LBMAL14]. This leads to ongoing research looking into proactive autoscalers [BHK17], [MBE13]. Proactive resource management needs predictions of the systems performance for different system configurations and workloads. Architectural performance models can provide these predictions. They model the systems architecture and its performance relevant characteristics, which are used to make predictions about the systems performance.
Different approaches for architectural performance models have been proposed focusing on different system attributes, modeling granularities and prediction approaches. Examples for architectural performance models include CACTOS [RIF01], SAMM [SAM], UML Marte [GS08], ACME [GMW10] and Palladio Component Model (PCM) [BKR09]. These models focus on predictions at design time, but for proactive resource management predictions at run time are required. The main differences are that at run time measurement values for the current system configuration are available and the existence of an upper limit on prediction time. A proactive autoscaler might need a result in thirty seconds or five minutes depending on the situation. Descartes Modeling Language (DML) [KBH14] is an architectural performance model targeting runtime scenarios. [HBK12] used DML to perform a case study on the influence of the resource environment on the systems performance, while [Bro14] evaluated the influence of different software and deployment aspects. DML also includes an adaptation model which allows the system to dynamically reconfigure itself to meet requirements specified in SLAs [HvHK+14]. For this purpose [HBS+16] proposes a control loop which uses DML predictions to reconfigure a system at runtime. This shows how DML can be used for autonomous proactive resource management. Proactive resource management depends on accurate predictions for the DML models [ZCB10].

For design time architectural models parametric dependencies were shown to increase prediction accuracy, especially for the exploration of different configurations and workloads [KHB06]. The performance of a component can depend on the value of a parameter which is passed from component to component. If a component is deployed in a different context the values of these parameters can change and therefore the component's predicted performance changes. In an online context information might not be available for all parameters. While the parameters are usually modeled, their distributions are not necessarily known. Some components might not be monitored due to performance reasons or because the parameter can not be measured. DML provides a parameterization model tailored to online scenarios which allows for incomplete parameter specifications. To facilitate model analysis, a single variable can be described using multiple dependencies. Dependencies can also be specified across components, instead of only within components. This allows to model dependencies between parameters on a coarser granularity. A parametric dependency might specify that the average number of web pages a user requests impacts the resource demand of a backend component. Formalisms except from DML limit dependencies to a chronological order. This means that only parameters contained in already executed behaviors can influence a variable. DML also allows for dependencies without chronological order, which provides additional ways to describe variables. In addition to the dependencies reaching across components this allows to specify dependencies that do not necessarily resemble real world influences but guesses about correlation. If only the database's resource demands can be measured, the performance engineer could specify a correlation between this resource demand an the resource demand of a frontend component. At the moment none of these parameterizations are taken into consideration during model solving and prediction, even though it would provide more accurate predictions.
The aim of this thesis is to provide a solution approach for parametric dependencies and apply it to DML. We provide a characterization for each model variable that depends on the path the request takes to encounter the variable. This description is independent from any parameters and can therefore be used for analytical and simulation-based stochastic performance analysis. In addition to basic parameter dependency features supported by existing work, we support multiple dependencies describing one model variable, dependencies that do not follow the call path, and dependencies spanning across components. To achieve the resolution of parametric dependencies we perform the following steps. We extract a `CallpathModel` representing all possible paths a request can take through the system. The `CallpathModel` details for every component to which other components the request can go. Then we transform the `CallpathModel` to the `RelationshipGraph`. This is a directed graph containing information about the models parameters and the dependencies between them. In this graph the nodes represent a parameter coupled with a path through the system to it. Every edge represents a dependency that can be used to derive a value for the edges target if the edges other parameters are known. Since the values for parameters can be arbitrary distributions we implement the computation of a distribution arithmetic. To resolve values for all nodes in the `RelationshipGraph` we use an adapted flooding algorithm [ADEP04]. From this solved `RelationshipGraph` a value for every parameter in every context can be retrieved. We integrate the information from the solved `RelationshipGraph` in the existing DML solving.
This allows to transform DML models with parametric dependencies to a Queueing Petri Net (QPN), which is subsequently simulated to provide performance metrics. Alternatively, other solving approaches like a transformation to Queueing Net (QN) or analytical solvers could be used. Models with parametric dependencies allow to model additional real life systems and the accuracy of existing models can be improved. Additionally, our approach provides two intermediate models, which provide additional views on the system. A look at the `CallpathModel` can help to understand the assembly of the system by displaying the paths a request can take through the system. The `RelationshipGraph` shows

the system's parameters and the dependencies between them. If not enough information is available to resolve a value for a parameter, the `RelationshipGraph` can be used to find out which measurements would allow for a full resolution. In addition to this, the `RelationshipGraph` provides an extension point for future features like the characterization of both model variables and dependencies, automated dependency learning and an automated decision support between multiple ways to characterize a parameter. While we implement the approach for DML, it can easily be applied to another performance model or any data center model that includes software control flows. To improve the transferability of our approach we provide a DML independent description of the approach. We provide an end-to-end evaluation of our approach using a media store model. To evaluate our approach we manually create reference models highlight specific modeling aspects and ti evaluate the approach end-to-end we use a media store model. To evaluate the distribution arithmetic we compare its results to empirically created samples.

Outline

The remainder of this thesis is organized as follows:

Chapter 2 details the foundations this master thesis is based on. This includes our definition of parametric dependencies, details on DML and how it models parameteric dependencies. Additionally it introduces Queueing Petri Nets (QPNs) and SimQPN, a simulation tool for the latter. Lastly, the current DML solving process which uses a transformation to QPN is described.

Chapter 3 describes the current state of the art with regard to modeling parametric dependencies. Additionally, the solving approaches for these models are introduced.

Chapter 4 contains the aim and respective approach of this thesis with regards to dependency modeling features and general modeling features of DML.

Chapter 5 discusses the resolution of parametric dependencies in general. It introduces the `CallpathModel`, the `RelationshipGraph` and an algorithm to solve the `RelationshipGraph`.

Chapter 6 details the adaptation of these ideas for DML, including DML specific versions of the `CallpathModel` and the `RelationshipGraph`. The extraction of these models is also covered in this section.

Chapter 7 features the technical details of the implementation, an overview over the components in our implementation and the control flow between them. Additionally, we describe its integration in the existing solver.

Chapter 8 evaluates the approach and its implementation. First we evaluate the calculation of a dependency by comparing its results with empirically generated samples. Next we used a number of manually created reference models to test if certain features are supported. Lastly we perform an end to end evaluation using a media store model.

Chapter 9 concludes the thesis with a short summary, an overview over the benefits gained from this theses and some ideas for future work.

2. Foundations

In this chapter we will introduces all the foundations needed to understand this master thesis. We first define the term parametric dependencies in Section 2.1, since this is the concept this master thesis is based upon. Next we introduce the architectural performance model DML in Section 2.2 and how it allows to model parametric dependencies Section 2.3. The current state of solving these parametric dependencies is detailed in Section 2.6. To understand the solving a rudimentary understanding of QPNs is necessary, therefore we give an introduction to QPNs in Section 2.4. These QPNs are solved using SimQPN which we introduce in Section 2.5.

2.1 Parametric Dependencies

Brosig [Bro14] gives a definition for parametric dependencies in models, which we will introduce in this section. When analyzing the performance of a system there are parameters that directly affect the systems performance like resource demands, response times, loop iteration counts or branch probabilities. But there are also parameters that do not directly correspond to the systems performance, that indirectly still influence the systems performance. Examples for this might be configuration values, component input parameters or the return values of calls to other components. While these parameters do not directly affect the systems performance, they instead influence the values of parameters that directly affect the systems performance. For example a components resource demand of a component might be influenced by the value of one of the components input parameters or the response time of a database query could depend on the number of entries in the relevant database tables. If we know all parameters that one variable is influenced by there exists an formula that describes its value. This formula would use the parameters that influence this variable as input. Aside from this formula a parametric dependency is defined by which value is characterized, the dependencies dependent and the parameters that influence this value, the independents. We will illustrate this concept using an example from [Bro14] which is shown in Figure 2.1. The first component, the `CatalogServlet` allows the user to browse a web shops articles. Every page shows ten articles and if the user wants to continue browsing he has to go to the second page. For each article a preview image is shown, which is retrieved from a database using the `JPAProvider` component. This component contains a cache which means already loaded images are retrieved faster. The probability of having to load the image instead of it already being in the cache can be modeled as a branching probability. This probability depends on the articles access

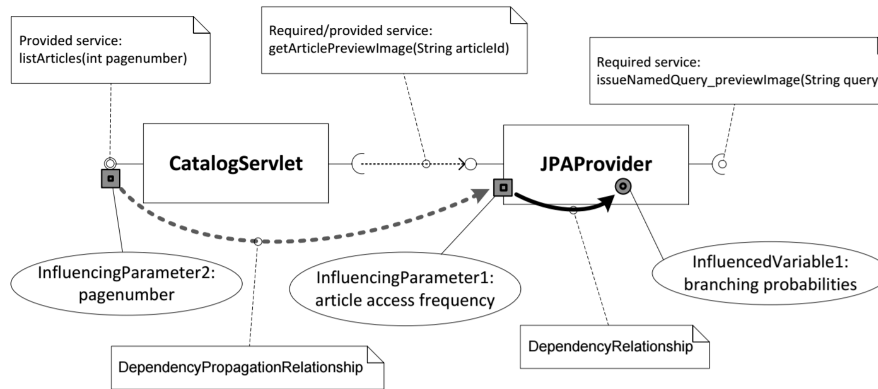


Figure 2.1: Example for parametric dependencies from [Bro14]

frequency and this frequency in turn depends on which page the user requests, since the later pages get called less often, so the article is accessed less often. So the `JPAProviders` branching probabilities can be calculated if the requested page numbers distribution is known.

2.2 Descartes Modeling Language (DML)

Todays software systems are becoming increasingly complex and dynamic while the individual parts are becoming more and more decoupled. This makes answering questions like the following difficult to answer. How would the system perform under a different workload? How much would the systems response time improve if we added 5 additional physical nodes to this cluster? These tasks are referred to as online performance prediction [KBHR10]. Descartes Modeling Language (DML) is a architectural performance model tailored for these scenarios. This means it models both the systems architecture and its performance relevant aspects. DML was used in a number of case studies [Bro14][HBS+16][HvHK+14][HBS+16]. The model is split into five sub-models: repository, assembly, resource environment, deployment and usage profile. The repository model defines blueprints for the 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 workload definition is contained in a separate model, the usage profile model. In the following we will go into detail about the individual models since they are critical to understanding this thesis.

Repository

Todays software systems consist of multiple independent components that can be flexibly assembled and deployed. Not every component contained in the system has to be deployed at all times and components will often times be deployed multiple times. To reflect this flexibility and to allow for reuse of single components DML separates the component descriptions from the information about the assembly and deployment of these components in the system. Components can not be defined independently from each other, since components can provide services that other components require to function. To define that component A requires component B would compromise the components independence. The classical component approach and DML deal with this by defining interfaces which can be provided by a component. If a component needs other components to function this is modeled by defining that the component requires an interface. This allows for every component that provides this interface to be used, for more details on this see Section 2.2. Figure 2.2 shows the interface meta-model of the DML repository model.

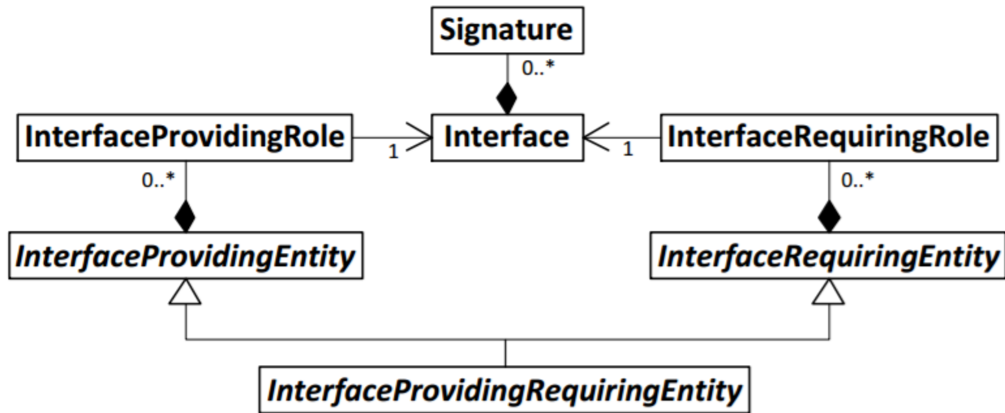


Figure 2.2: Interface meta-model from [Bro14]

Every `Interface` defines a series of `Signatures` which represent the different functionalities encapsulated in an `Interface`. If a model element can provide interfaces it is derived from `InterfaceProvidingEntity`, which contains a number of `InterfaceProvidingRoles` which each reference one `Interface`. If a `InterfaceProvidingEntity` would provide multiple `Interfaces` it would contain multiple `InterfaceProvidingRoles`. The `InterfaceRequiringEntity` and `InterfaceRequiringRoles` provide the ability to require `Interfaces` using the same mechanisms. If an element can both provide and require `Interfaces` it is modeled as an `InterfaceProvidingRequiringEntity`. The `InterfaceProvidingRequiringEntities` in DML are shown in Figure 2.3.

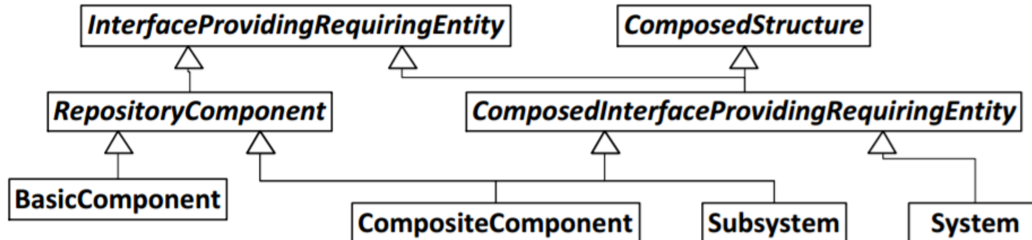


Figure 2.3: InterfaceProvidingRequiringEntity meta-model from [Bro14]

A traditional component is one example for an `InterfaceProvidingRequiringEntity`, which is modeled in DML as a `BasicComponent`. A `CompositeComponent` is a collection of `RepositoryComponents` that together form a larger component that can again require and provide `Interfaces`. The `System` and `Subsystem` are elements from the assembly model which is explained in Section 2.2. Aside from being able to provide and require `Interface` every `RepositoryComponent` contains `ServiceBehaviorAbstracts` which describe the performance relevant aspects of each `Signature` of every `Interface` provided by the component. This means it contains a `ServiceBehaviorAbstraction` for every functionality which the component provides. The most common `ServiceBehaviorAbstraction`, the `FineGrainedBehavior` is shown in Figure 2.4. Every `FineGrainedBehavior` contains exactly one `ComponentInternalBehavior`, which is a ordered list of `AbstractActions` which represent different steps of the behavior. The two basic `AbstractActions` are the `InternalAction` which wraps a `ResourceDemand` and the `ExternalCallAction` which models a components calls to its `InterfaceRequiringRoles`. The remaining `AbstractActions` are control flow actions, which allow to model loops, forks and branches. This behavior model together with the `InterfaceRequiringRoles` and `InterfaceProvidingRoles` allows to

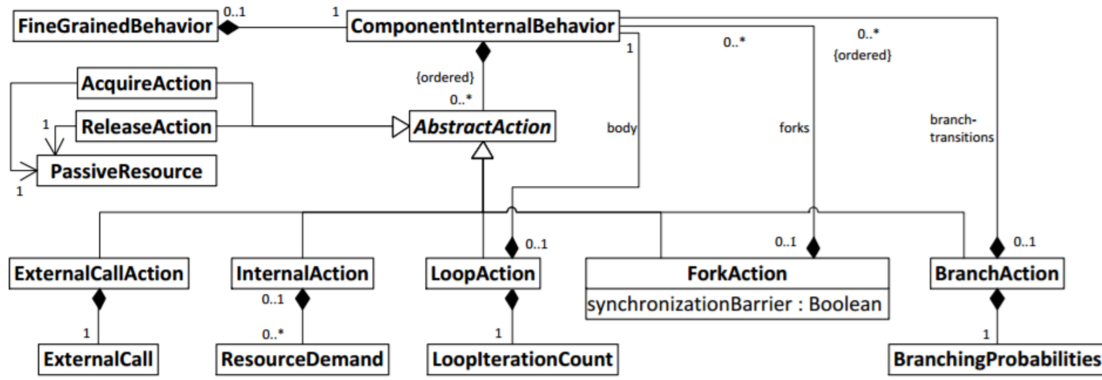


Figure 2.4: Behavior meta-model from [Bro14]

model both the systems architecture and its performance attributes.

Assembly

In the assembly model users can specify a system using the components from the repository, that is either analyzed directly or reused in a larger system. An Unified Modeling Language (UML)-Diagramm for the meta-model of the assembly is shown in Figure 2.5.

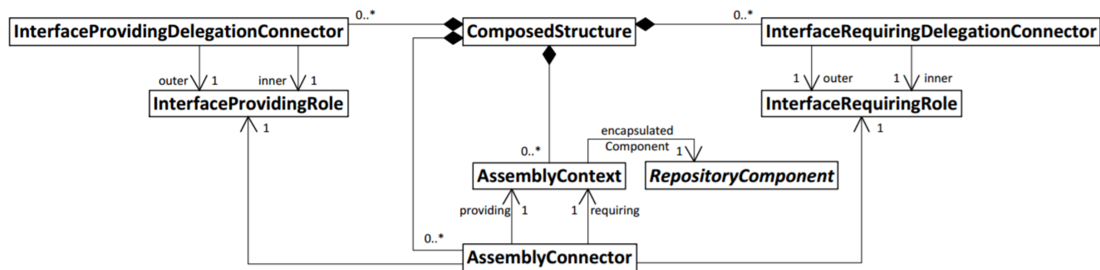


Figure 2.5: Assembly meta-model from [Bro14]

The head element for a assembly is the `ComposedStructure`, which uses the `Interface-ProvidingDelegationConnector` to propagate the systems `InterfaceProvidingRoles` across they system border. Should the system require additional services to function these `InterfaceRequiringRoles` are propagated using the `InterfaceRequiringDelegation-Connectors`. Inside the system the `RepositoryComponents` from the repository model are encapsulated by an `AssemblyContext` to represent an instance of the component. As described previously the encapsulated `RepositoryComponents` have their own `Inter-faceProvidingRoles` and `InterfaceRequiringRoles`. To connect a `InterfaceRequir-ingRole` with a `InterfaceProvidingRole` for the same interface the assembly model uses `AssemblyConnectors`. For the model to be valid all `InterfaceRequiringRoles` inside the system have to be connected to corresponding `InterfaceProvidingRoles`.

Resource Landscape

The physical resources of the system are modeled in the resource landscape model. It contains information about the active and passive resources of the system as well as their organization in different data centers and composite hardware resources like racks. This organization is shown in Figure 2.6.

A `DistributedDataCenter` consists of multiple `Datacenters`, that each contain a number

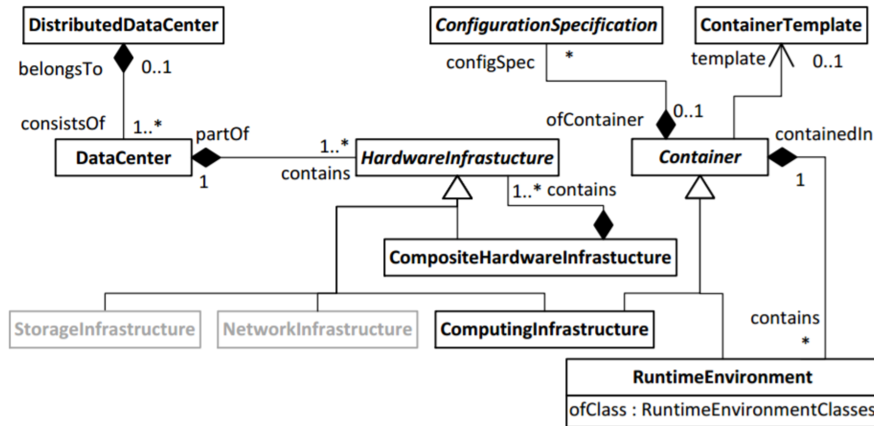


Figure 2.6: Resource Enviroment meta-model from [Bro14]

of `HardwareInfrastructure`. A `HardwareInfrastructure` can be a `CompositeHardwareInfrastructure` that itself consists of multiple `HardwareInfrastructures`. The other type of `HardwareInfrastructures` are the `ComutingInfrastructers`, that can either contain a `ConfigurationSpecification` or a `RuntimeEnviroment` that in turn contains a `ConfigurationSpecification`. A `RuntimeEnviroment` represents a virtual machine that either contains additional virtual machines or a hardware resource modeled by a `ConfigurationSpecification`. A detailed UML diagram for the `ConfigurationSpecification` is shown in Figure 2.7.

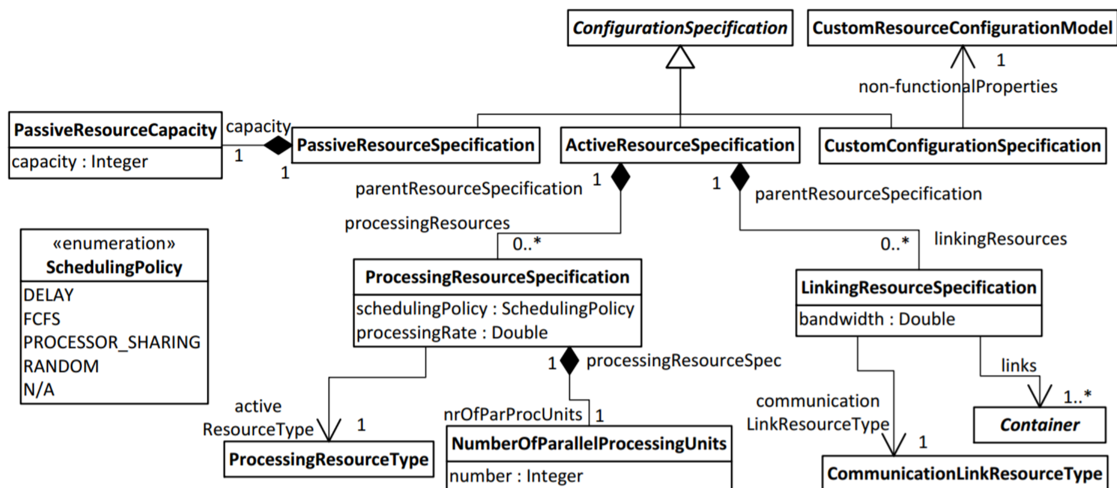


Figure 2.7: Resource Specification meta-model from [Bro14]

Since the details of the resource specifications are not relevant for this thesis we will not go into much detail here. It should be noted that a resource specification can either be a `ActiveResourceSpecification` or a `PassiveResourceSpecification`. A passive resource could be a thread pool or a limited number of database connections. Active resources represent processing power like a CPU or a HDD, which is modeled using a processing rate and a `NumberOfParrallelProcessingUnits`. Together this allows to model a detailed representation of a data center's physical landscape.

Deployment

The deployment model links the assembly and the resource landscape models by describing

which component instances are deployed on which physical or virtual machines. This is done using the meta-model described in Figure 2.8.

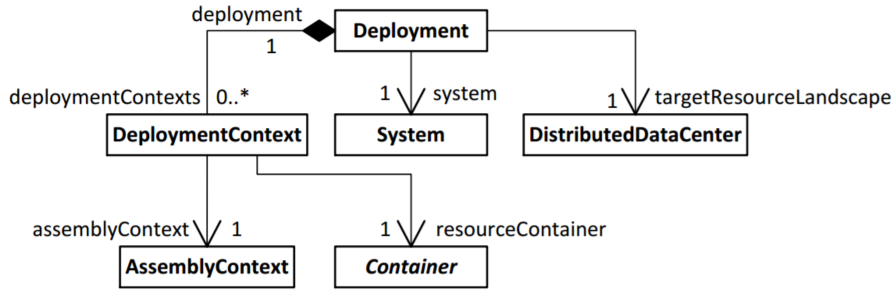


Figure 2.8: Deployment meta-model from [Bro14]

Each `Deployment` model links exactly one `DistributedDataCenter` and one `System`, the root nodes of the resource landscape and the assembly respectivly. A `Deployment` consists of a number of `DeploymentContexts` that each link one `AssemblyContext` from the assembly with one `Container` from the resource landscape.

Usage Profile

The workload is modeled independently of the rest of the DML model to simplify analysis on a system with different workloads. Each `UsageProfileModel` references exactly one `System` and therefore assembly model as shown in Figure 2.9.

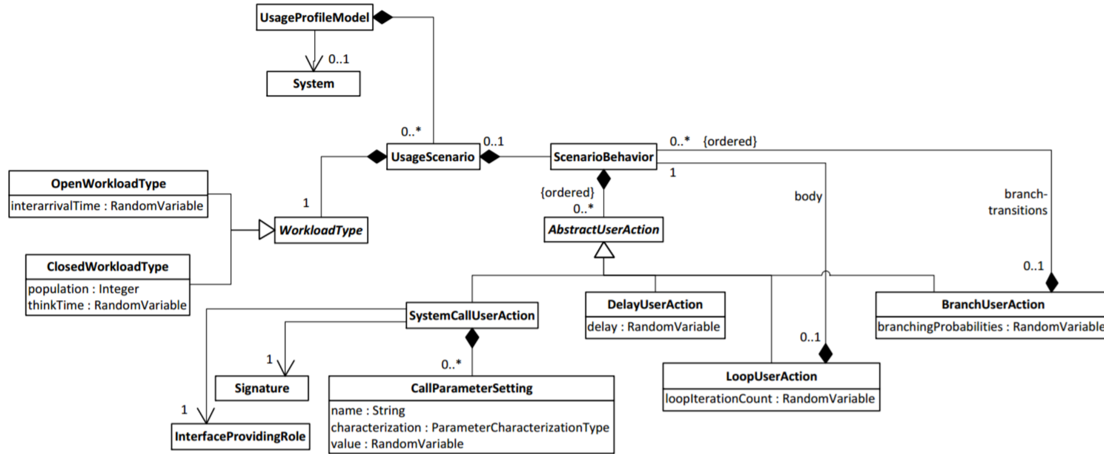


Figure 2.9: Usage profile meta-model from [Bro14]

Each `UsageProfileModel` specifies one or more `UsageScenarios` that describe a way users access the system. These `UsageScenarios` are used to model simultaneous access from different workload types. Each `UsageScenario` has either a `OpenWorkloadType` defined by its inter arrival rate or a `ClosedWorkloadType` consisting of a population and a think time. These `WorkloadTypes` describe how often the `UsageScenario's ScenarioBehavior` is executed. A `ScenarioBehavior` consits of a order list of `AbstractUserActions` that are executed in order of the list. The most common `AbstractUserAction` is the `SystemCallUserAction` where a `Signature` of a `InterfaceProvidingRole` of the `System` is being called. To model a delay between two `SystemCallUserActions` the `DelayUserAction` is used. Both loops and branches can be modled inside the `ScenarioBehavior` using

the `LoopUserAction` and the `BranchUserAction`. Together this allows to flexibly model different types of workloads.

2.3 Parametric Dependencies in DML

In this section we will detail how DML models dependencies, but to do this we will first need to go into detail about what kind of variables and parameters exist, that can get connected using dependencies. DML's variables, which are all derived from the `ModelVariable` class are shown in Figure 2.10.

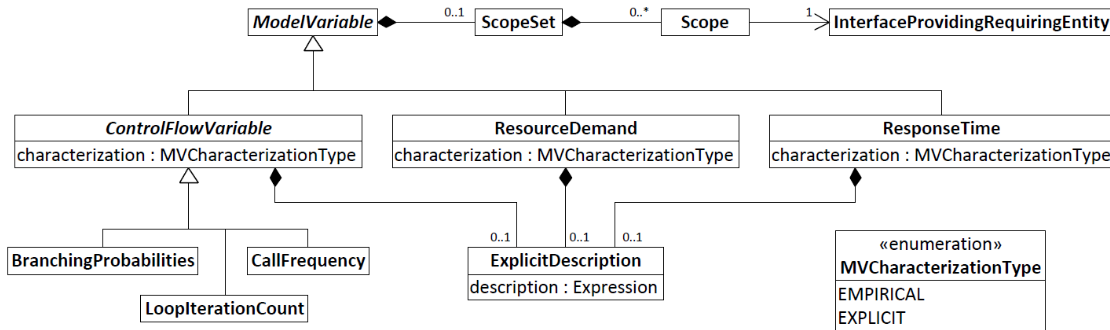
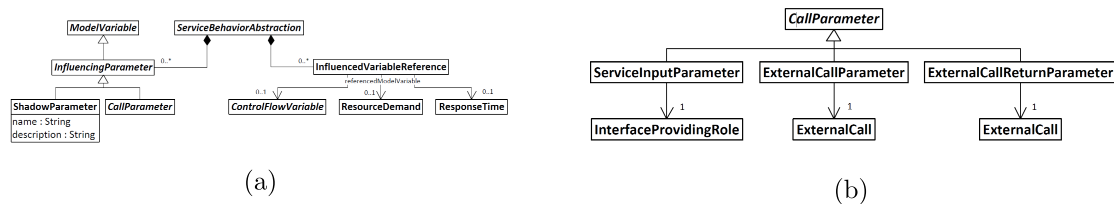


Figure 2.10: Variables meta-model from [Bro14]

There exist five different `ModelVariables`, `BranchingProbabilities`, `LoopIterationCount`, `CallFrquency`, `ResourceDemand` and `ResponseTime`. Each of these variables can be characterized either as `EMPIRICAL` or `EXPLICIT`. `EXPLICIT` variables are characterized using an `ExplicitDescription`, while `EMPIRICAL` variables are characterized at run time using measurement values. To describe these `ModelVariables` using a dependency they have to wrapped in an `InfluencedVariableReference` as shown in Figure 2.11 (a). These a `InfluencedVariableReferences` can depend on `InfluencingParameters`. The first type of `InfluencingParameter` is the `ShadowParameter` which is used to model influencing parameter that do not directly map to a component property. The other `InfluencingParameters` are the `Callparameters`, which are shown in Figure 2.11 (b). A components input parameters are modeled as `ServiceInputParameters`, while the parameters of an `ExternalCall` are modeled as `ExternalCallParameters` and `ExternalCallReturnParameters`. The `InfluencedVariableReferences` and the `InfluencingParameters` can be used in relationships as described in Figure 2.12. A `DependencyRelationship` describes an `InfluencedVariableReference` using one or more `InfluencingParameters`. To describe relationships between parameters on an assembly level `DependencyPropatgationRelationships` can be used. They describe an `InfluencingParameter` using one more `InfluencingParameters`. The `ComponentInstanceReferences` are used to describe in which assembled components these parameters exist. A Relationship can be either explicitly described or empirically characterized at run time.



(a) (b)

Figure 2.11: Parameter meta-model from [Bro14]

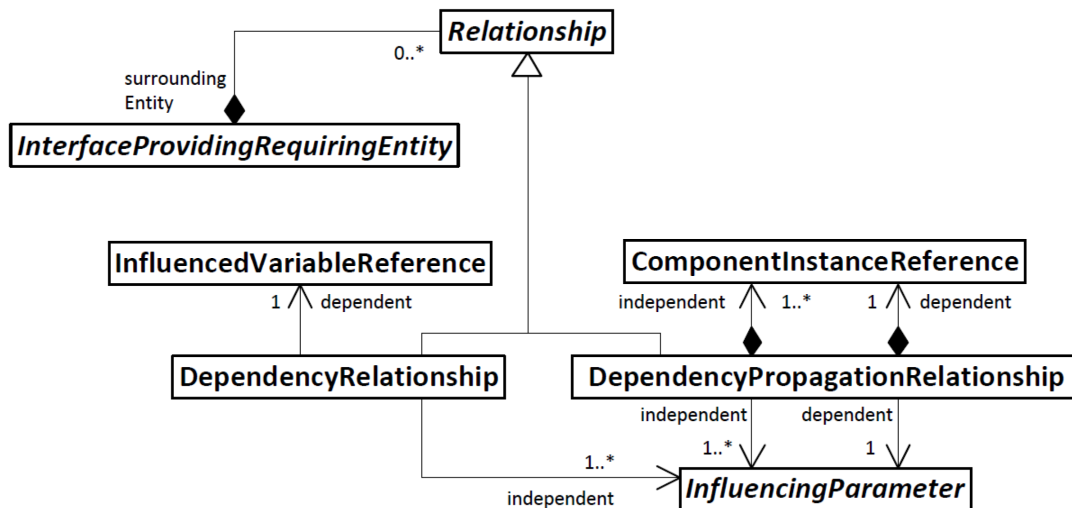


Figure 2.12: Relationship meta-model from [Bro14]

2.4 Queuing Petri Net (QPN)

The Queueing Petri Net (QPN) is a formalism from the area of Queuing Theory. It is a combination of two prior formalisms, the Queueing Net (QN) and Petri Net (PN). It consists of four elements, Places, Tokens, Transitions and Queuing Places, which are shown in Figure 2.13. The places have no functionality by them selfs, but they can contain

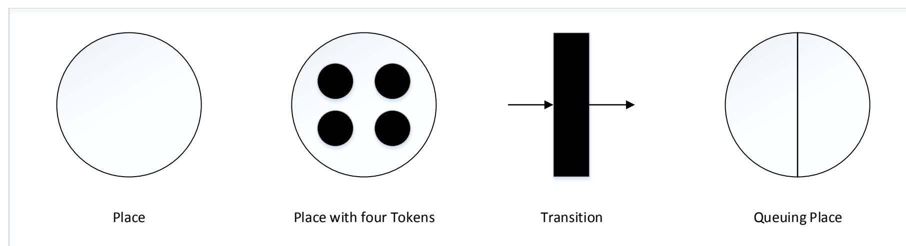


Figure 2.13: QPN Formalism

tokens. These tokens are the only element that changes during a QPNs life, facilitated by the transitions. These transitions connect one or more places and move tokens between them. They fire when all required tokens, denoted by the incoming arrows are available. Upon the activation of a transition all required tokens are consumed and the output tokens, denoted by the outgoing arrows are created. So far the formalism is similar to PNs, the real difference are the newly added queuing places, which are borrowed from the QNs. Just like ordinary places queuing places can be connected via transitions. Incoming tokens arrive at the left side of the queuing place, the so called waiting area. From there on one token at a time can be dealt with by the so called server, the order in which the tokens enter is usually first come first served. After a set amount of time the token is moved to the right side of the queuing place where it is available for transitions. These elements are used to represent delays of any sort, for more information on how a QPN can be used to represent a software environment see Section 2.6.

2.5 SimQPN

SimQPN is a simulator for QPNs. It facilitates a discrete-event-simulation approach that allows to simulate larger amounts of simulated time in the same period of real time than

a traditional simulation. It supports three different analysis methods, the batch means method [MS84], replication/deletion approach [Law15] and the method of welch[Wel83]. Currently the warm-up period of the simulation has to be manually specified, but there is some ongoing for the implementation of an automated detection for the warm-up period length. SimQPN allows the user to choose between three different stopping criteria for the simulation, with the simplest one terminating the simulation after a fixed time. The simulation can also be stopped when a relative or absolute precision is reached. The time between these stop checks can be set in both real time and simulated time. A statlevel that that indicates what level of detail the collected information should have can be set for every place in the QPN. These statlevels influence the duration of the simulation since the collection of aggregates is faster than collecting all values. For more information about SimQPN see [KSM10].

2.6 DML Solution Process

In this section we detail how DML is currently solved using QPNs. We start in Section 2.6 by explaining the stackframe model that is used during the transformation to QPN. How DML is currently transformed to the stackframe model is detailed in Section 2.6. The stackframe model is then transformed to a QPN as described in Section 2.6. Finally the resulting QPN is solved using simQPN (see Section 2.5).

Stackframe model

The stackframe model contains information about the paths a request can take through the system, as well as characterizations for the variables a request encounters during its traversal of the model. These characterizations can come from empirical characterizations, from relationships or explicitly stated within the model. It also describes which `ServiceBehaviorAbstraction` should be used to solve the model. While it contains information about which `ServiceBehaviorAbstraction` the user calls, it does not describe the workload. The physical aspects of the model are also not considered. So together with the `UsageProfile` and the `Deployment` it builds the basis for any model based solver. This avoids duplicate code between multiple model based approaches, since they do not each have to solve model traversal and variable characterization.

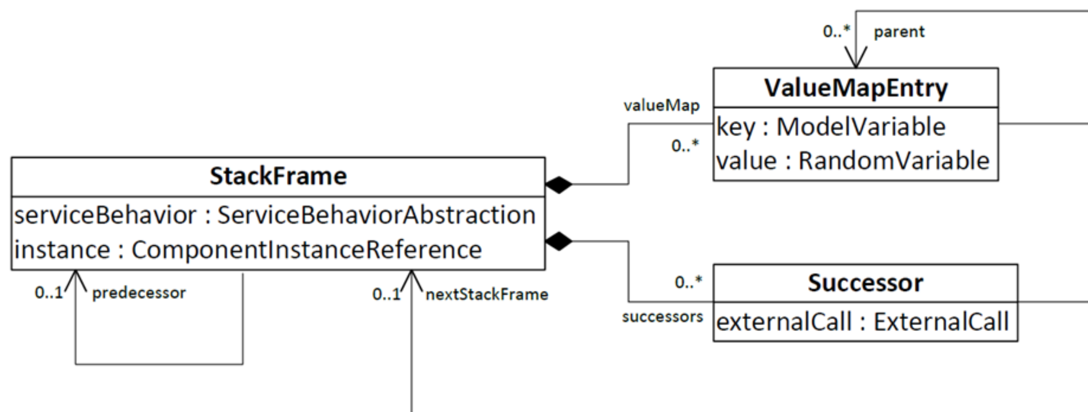


Figure 2.14: Stackframe Metamodel

Figure 2.14 describes the meta model for the stackframe model. Each `Stackframe` links a `ServiceBehaviorAbstraction` to a `ComponentInstanceReference`. This `ComponentInstanceReference` is the entry point to use the `Deployment` to extract information about

hardware aspects of the model. For each `ModelVariable` in the `ServiceBehaviorAbstraction` there is an `ValueMapEntry` that contains its characterization in form of a `RandomVariable`. A `Stackframe` contains a `Successor` for each `ExternalCall` in the `ServiceBehaviorAbstraction`. The `Successor` describes which `Stackframe` the `ExternalCall` calls.

Transformation to Stackframe Model

The current implementation of the transformation does not take any relationships or empirical variable characterizations into account. It only traverses the model and extracts the possible successors for each `ServiceBehaviorAbstraction` alongside the explicit variable characterizations. [Bro14] describes a more detailed transformation that is not implemented.

Transformation to QPN

The transformation to QPN does not transform the model as a whole, but it describes modularized mappings for each DML element, that together allow to transform the whole model. Most of these mappings were first described in [MKK10]. Each mapping has an ordinary place that is used as an entry point for the structure and another ordinary place that is used as an exit point. Mappings for elements like `ExternalCall` that call other elements also have entry/exit points for the elements that are called. By linking these entry/exit points for the external call to the entry/exit points for the called element the modular mappings come together as a QPN that models the whole DML model. In the following we explain the mappings for `ClosedWorkloads`, `ExternalCalls` and `Acquire-/ReleaseActions`, for the remaining mappings see [Bro14].

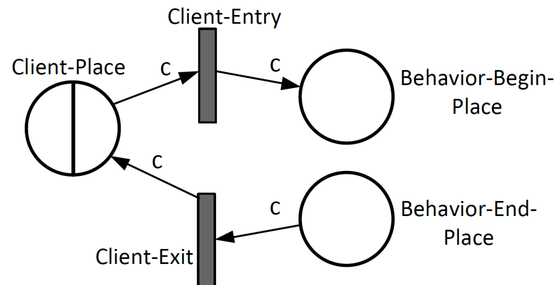


Figure 2.15: Mapping for ClosedWorkload [MKK10]

A `ClosedWorkload` mean a limited number of requests exists and after they are finished by the system they have to wait a certain time before the are processed by the system again. To model this we have a queuing place that represents the wait time, after a request leaves the queue it enters the begin place of the called behavior. Upon leaving the behavior end place the request enters the queue again.

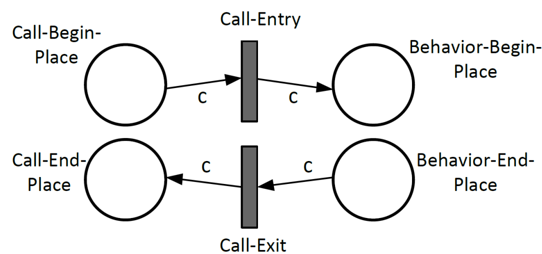


Figure 2.16: Mapping for ExternalCall [MKK10]

To model an `ExternalCall` we need to stop the current behaviors execution and wait until a external behavior is done. To model this our token leave the current behavior and enters the behavior begin place and upon returning from the behavior end place the `ExternalCall` ends.

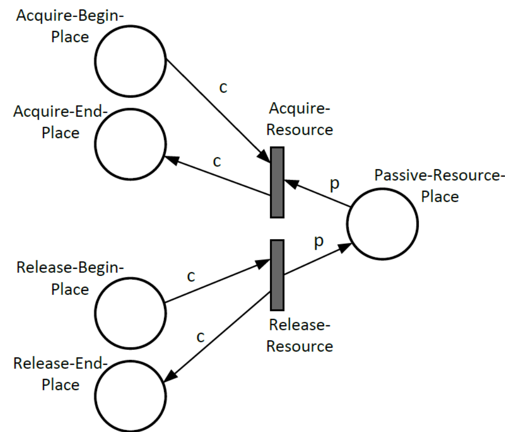


Figure 2.17: Mapping for Aquire-/ReleaseAction [MKK10]

An `Acquire-/ReleaseAction` means that to execute a behavior the additional constraint must be met that not more than a set number of request is currently in the area covered by the acquire/release. We model an `AcquireAction` by a transition between the acquire start place and the acquire end place that requires a token from a passive resource place. The corresponding release action is modeled by a transition that places a new token in the passive resource place. This means that only a set number of requests can be executed concurrently.

3. Related Work

There has been a lot of work done already in the area of performance modeling and there exists to model and resolve parametric dependencies, that we all found unfit to use in the case of DML. We will detail to best of our knowledge what has been done so far and what has not yet been done. In Section 3.1 we describe which performance models implement parametric dependencies and if so how they model them. In Section 3.2 we briefly go over the idea of extracting performance models that contain parametric dependencies using measurement data. How these performance models resolve their version of parametric dependencies is described in Section 3.3. Some performance models allow for operations between multiple continuous distributions inside their dependencies. The current state of the art for techniques to calculate and compute these operations between continuous distributions is also described in Section 3.3. Together this describes the current state of the art, which we build upon in this master thesis.

3.1 Modeling of Parametric Dependencies

In [GM01], [GS08], [Zsc10], [BdWCM05], [Phi], [Bro14] and [BHK12] different modeling approaches for parametric dependencies are proposed. Every approach has its own advantages and disadvantages but none of these approaches target an online scenario except for DML [Bro14]. In the following we provide an overview over the different meta-models focusing on their approach to model parametric dependencies. We go from the model with the least detailed parametric dependencies to the most detailed model. In [GM01] software components are modeled using UML and are extended with annotations about the performance relevant parameters. This model is then mapped to a QPN where it can be simulated to predict response times. However, they do not model dependencies between the parameters. Instead the parameters are inserted directly in the resulting QPN since they represent resource demands or loop frequencies. Another approach modeling parametric dependencies is UML MARTE [GS08] which adds capabilities to UML for model-driven development of real-time and embedded systems (RTES). It allows for the modeling of non-functional properties which can also include performance properties. In MARTE these non-functional parameters are used in constraints that need to be satisfied. A formal description for non-functional attributes for components has been proposed in [Zsc10]. These non-functional attributes can be used to model performance relevant parameters. This description allows to model a components performance in relation to the available resources or the container it is contained in, not in relation to the components

input parameters. [BdWCM05] extends ROBOCOB [Phi], an existing performance model to support input parameters. Each component can have one or more input parameters which are used in the cost function that describes the components resource demand. Neglecting output or return parameters simplifies the solving significantly but leads to not being able to model some systems adequatly.
In Palladio Component Model (PCM) [Koz08] each component can have input parameters and output parameters. If a component specifies a call to another component, the call itself has input and output parameters which are propagated between the components. The component developer can now define dependencies between these parameters, for example an external calls input parameter could be two times the components input parameter. These dependencies can also be specified between parameters and model variables like resource demands or loop iteration counts. This allows to define non-static components that behave differently depending on where it is called from and also depending on which implementation of an interface it calls.
DML [BHK12] extends the modeling concepts from PCM to support online scenario in a number of ways. DML allows for model variables to be described by multiple dependencies. A resource demand could be characterized by two dependencies. Therefore the influencing parameters of one dependency need to be known to resolve a value for the resource demand. So the constraint that all parameters need to be known is no longer a concern. Additionally, DML supports modeling of dependencies on an assembly level which means they can span across multiple components. For example, a components resource demand can directly depend on another components input parameter. This simplifies modeling and improves understandability at the cost of the components reusability. Both model variables and relationships can be modeled as empirically characterized, which mean the values should be characterized using measurement values from the system. To our best knowledge there have been no attempts to model parametric dependencies in low level models like QPNs or stochastic process models. This is hard to implement since parametric dependencies need high level information, which is not available in low level modeling formalisms.

3.2 Extraction of Parametric Dependencies

Modeling dependencies manually is realistic for smaller systems, but for enterprise level systems modeling them by hand is time intensive. Large systems usually employ some kind of response time monitoring. Example Application Monitoring Tool (APM) tools include DynaTrace [WBGK15] and Appdynamics [WH13]. Kieker [VHWH12] is an APM tool from research. These measurements can be used to extract performance models [Bre16], [BK17], but most of these approaches do not extract parametric dependencies. An approach that does extract parametric dependencies was proposed in [KKR10], which uses bytecode analysis to extract PCM models for software components that contain parametric dependencies. They analyze bytecode in combination with measurements for parameter values at interface level and then employ genetic search to find correlations between the two. They then map certain bytecode instructions to model variables like loop iteration counts. Therefore they can derive characterizations for these model variables based on the input parameters of the component and the output parameters of components called from this component. An approach based purely on measurement values is proposed in [BHK11]. The proposed approach does not try to actively learn between which parameters dependencies exist. Instead it relies on the user to specify where dependencies exist and the characterization of the dependency is learned from measurement values. This prevents both learning wrong dependencies and learning correlations, that are small enough to be potentially from random noise. It comes with the downside of needing more input from the user. How these models can be solved to enable predictions is described in Section 3.3.

3.3 Resolution and Calculation of Parametric Dependencies

Resolving a parametric dependency means to provide a description for the parameters that depend on other parameters that no longer contains these other parameters. In the context of performance modeling this usually means to provide a value for each possible scenario which leads to a component behaviors execution. In the following we discuss solution strategies for solving of parametric dependencies based on formalisms presented in Section 3.1. For [GM01], [GS08] and [Zsc10] there are no parallels to DML dependencies, while [BdWCM05] and [Koz08] could be relevant for the DML dependency resolution. In [GM01] the parameters are directly copied into the QPN so no resolution is necessary. Both [GS08] and [Zsc10] do not propose a solving algorithm, but they both also do not use fully fledged parametric dependencies. [BdWCM05] models only input parameters so they use a flooding algorithm to resolve values for all parameters. This works by starting at the workload definition where the input, parameters need to be known and from there these input parameters can be propagated to all components that get called. If these components contain calls to other components these input parameters can again be propagated to the next component. This is done until a component is reached where the call terminates, which means it does not contain calls to any other components. Visually this can be imagined as the parameters starting in the workload definition and then flowing through the systems components.

We go into detail on the solving approach used for parametric dependencies in PCM, since parameteric dependencies in DML are an extension of Palladio Component Model (PCM) approach to model parameteric dependencies. Koziolek [Koz08] introduces the concept of ComputedUsageContexts to simplify the dependency resolution. A ComputedUsageContext is a wrapper for a behavior description and an assembly instance of the component. Additionally it stores the behaviors input and output parameters as well as the in- and output parameters for each external call inside the behavior. Two ComputedUsageContexts are considered equal if the behavior description, the assembly instance and the input parameters are identical. To resolve the dependencies the model is traversed and for each encountered behavior description the corresponding ComputedUsageContext is retrieved or a new one is created if it does not exist yet. The traversal starts by going over each behavior that is called from the usage profiles calls to the system. The dependency solver goes over each action contained in the behavior and if an external call is encountered its target is traversed first before returning to the next action inside the current behavior description. When creating the ComputedUsageContext for the target of an external call the current ComputedUsageContext's external call parameters are mapped to the new ComputedUsageContext's input parameters. Similarly after returning to a ComputedUsageContext from traversing an external call the target ComputedUsageContext's output is mapped to the current ComputedUsageContext's external call output. If any action encountered during the model traversal contains a model variable that is characterized using a dependency the current ComputedUsageContext's parameter values are used to calculated the model variables value. At this point all parameters that are used to describe the model variable have to be known due to the constraints described in Section 3.1. After traversing the model a characterization for each model variable is known for every set of occurring input parameters for the containing behavior description.

Parameters in DML can have different data types like boolean value, integer, probability mass function or probability distribution function. Calculating the resulting value for a dependency can contain arithmetic operations like addition, subtraction, multiplication and division or comparisons like greater or equal. Most of these operations are straightforward to compute, but the operations between two distributions can be challenging. Therefore, we will give a small overview over the current state of the art of addition, subtraction, product and division of distributions. The addition of two distributions is also known as the convolution of two distributions. Formulas for the convolution can be found in fun-

damental statistics literature like [Sch12]. The convolution also covers the subtraction of two distributions since we can inverse the second distribution and add both. A formula for the product of two distributions has been introduced in [ST66]. An approach for the division of two distributions has been proposed in [Cur41]. While deriving formulas for arithmetic operations on distribution is possible, the computation proves difficult since it involves the solution of complex integrals [GLD04]. [GLD04] also proposes an approach based on the melin transformation to compute the product of two distributions. Solutions for the products of Beta, Gamma, and Gaussian random variables have been introduced in [ST70]. [Akk08] describes the convolution of exponential distributions. While for some cases solutions exist, a general solution does not exist.

4. Approach

The goals of this master thesis are described in Section 4.1 and the approach to achieve these goals in Section 4.2.

4.1 Goals

The three main aims of this master thesis are the determination of the correct dependency resolution order, the calculation of the dependency results and the integration of the new transformation to the `Stackframe Model` in the existing QPN based DML solver.

1. **Dependency Resolution** The order in which the dependencies are resolved is essential since a variable which was characterized using one dependency might be used as an input parameter in another dependency. Resolving them in reverse order is impossible. To determine the correct order an overview of the variables and dependencies in the system is needed. We differentiate three different types of dependencies in DML. `DependencyRelationships` which describe dependencies within one component, `DependencyPropagationRelationships` which describe dependencies across component borders and implicit relationships which are not explicitly modeled.

    a) **Component Relationships** `DependencyRelationships` model dependencies with the component's borders. A resource demand can depend on an input parameter or on the value returned by an external call. These relationships can not be resolved independently since the component's input parameters as well as the values returned by a call to another component depend on the remaining system.

    b) **Implicit Relationships** If two parameters model the same real world value due to the system's assembly, they have to be equal. This constraint can be understood as an implicit relationship. The input parameters of an external call have to be equal to the input parameters of the called component. Alternatively, if a component returns a value this value has to be equal to return parameters of the external call that lead to its execution.

    c) **Assembly Relationships** In DML the `DependencyPropagationRelationships` allow to model dependencies not on a component basis, but between assembled components. This provides challenges for the dependency resolution since other components apart from those directly connected to the resolution target are of relevance.

2. **Dependency Calculation** DML allows for boolean values, discrete distributions and continuous distributions to be used as values for a parameter or model variable. These values can be used as input for boolean operations, arithmetic operations and comparisons in the explicit description of an relationship. We implement an algorithm to compute solutions for these operations.
   a) **Boolean Calculations** We support boolean operations like AND, OR and XOR for both boolean values and probabilistic boolean values.
   b) **Discrete Numeric Calculations** For discrete numeric distributions support both arithmetic operations like +,-,*,/ and comparisons like <,=,>, ≥, ≤.
   c) **Mixed Numeric Calculations** For a mix of continuous and discrete operations to support as many of the previously mentioned operations as possible.
   d) **Continuous Numeric Calculations** For two continuous distributions we support all mentioned arithmetic operations and comparisons, except for divisions since the computation of the division of two continuous distributions is unsolved so far, as mentioned in Chapter 2.
3. **Integration** We integrate our approach in the existing DML tool chain. Using the values derived from the relationships as input for the existing transformation to QPN which is then be solved using SimQPN. This results in an automated process to solve models containing relationships and allow users to specify Descartes Query Language (DQL) performance queries to answer their performance relevant questions. Furthermore, this enables end-to-end evaluation of our approach.

While features might be evaluated separately, combinations of modeling features cause additional challenges. These scenarios have to be considered for our approach. Supported features as well as their respective evaluation are listed below:

1. **Relationship Chains** If the `InfluencingParameters` of a relationship are not explicitly modeled the relationship might still be resolved through values retrieved from other relationships necessary for the unknown `InfluencingParameters`. The `InfluencingParameters` of these relationships might in turn be not explicitly modeled but again the dependent of a relationship. This forms chains of relationships that have to be resolved in order. If an independent of a relationship in not yet characterized, a relationship with this parameter as a dependent needs to be identified and solved first.
2. **Multiple Callpaths** The resource demand of a component might be different depending on which component it is called from. The resource demand depends on the components input parameters, which can be different for two external calls. But due to relationship chains the resource demand might depend on the input parameter of a completely different component. Thus, not only the component the call originated from, but the whole path of the request through the system needs to be taken into account.
3. **Multiple Instances of one component** For a component which is instantiated multiple times, independent values for the contained model variables have to be provided by the dependency solving. Depending on the call path, these model variables can have different values despite being contained in the same component instance. The two identical components can be on different call paths and therefore have different input parameters.
4. **Multiple ServiceBehaviorAbstractions** A component providing `ServiceBehaviorAbstractions` for multiple signatures should not interfere with the dependency

resolution. Despite being a technical more than a conceptual challenge this problem needs to be taken into consideration.

4.2 Approach

In this section we will give a short overview over our approach to resolve parametric dependencies. The approach can be split into three steps:

1. **Extract callpath information** Parameters can have different values depending on where the containing component is called from. In a model this is realized by different dependencies describing them or because parameters they depend up having different values when called from another component. Therefore to analyze the models dependencies the paths a request to the system need to be taken into account.
2. **Extract parameter and dependency information** On the basis of the callpaths in the model, detailed information about the model's parameters and dependencies can be extracted. For every pair of parameter and callpath leading to it is described which other parameter and callpath pairs it depends upon. This is coupled with a description what equation can be used to calculate a value for the parameter using the values of the independent parameters.
3. **Resolve all possible dependencies** Having the information about parameters and their dependencies in a comprehensive format we contribute an algorithm to resolve value for all parameter and callpath pairs. Should resolving a value for some parameters be impossible due to missing information the algorithm resolves all values that can be calculated.

The resolved values for parameters can be used to either simulate the model or to use an analytical approach to solve it. Additionally, step two provides an overview of the models parameters and dependencies. On the one hand this can help the user to understand the model better and on the other hand this provides an extension point to implement loading of empirical variables, automated learning of dependencies and empirical characterization of dependencies. Additionally it provides an opportunity to implement more sophisticated decision support between different ways to characterize a parameter.

5. Solving Parametric Dependencies

This chapter describes our proposed approach for the resolution of parametric dependencies. First all possible paths through the system are extracted to a `CallpathModel`, as described in Section 5.1. In Section 5.2 we use this `CallpathModel` to extract a `DependencyGraph` containing all variables and their dependencies. How this `RelationshipGraph` can be solved is explained in Section 5.3. The calculations necessary to calculate values from a dependencies where all influencing values are known are detailed in Section 5.4.

5.1 Callpath Model

Assigning a single value to a components resource demand is often not feasible, because it depends on some input parmeter or an external calls output parameter. And therefore by extension it depends on from where the current component is called and which implementation fulfills its required components.This means we need information about which paths a request can possibly take through the system. We propose to extract this information to a `CallpathModel`, an example for which is shown in Figure 5.1.

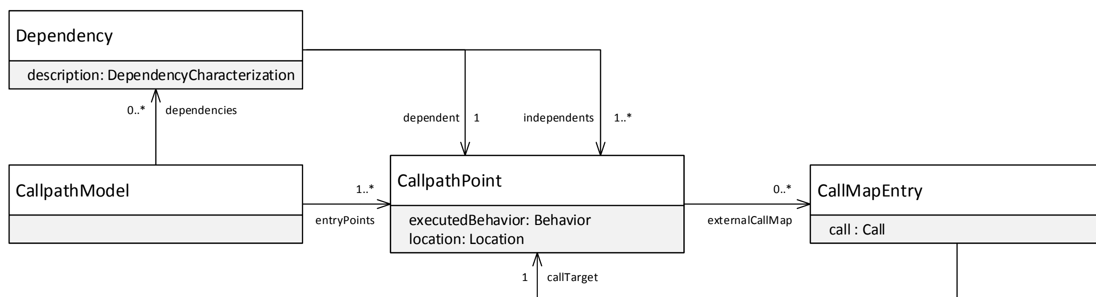


Figure 5.1: CallpathModel Metamodel

The `CallpathModel` contains a set of `CallpathPoints` that represent possible entry points for the system. Each `CallpathPoint` specifies the `Location` where a `Behavior` is executed. A location does not refer to a physical node in the system, but to an assembled component that contains the executed behavior. Every time the `CallpathPoint's Behavior` specifies a `Call` to another component a `CallMapEntry` references the `CallpathPoint` that represents the `Behavior` that is executed due the the `Call`. This model leads to a tree-like

structure as shown in Figure 5.2. Every possible path that leads to a `Behaviors` execution can be described by a single `CallpathPoint`, since the path to it can be traced back due to it being contained in a `CallMapEntry`.

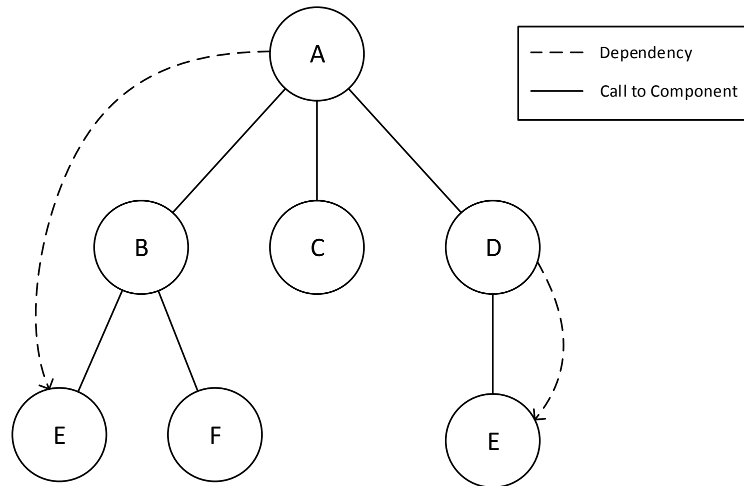


Figure 5.2: Example for a callpath model

When specifying `Dependencies` in this model it is only important in which components the dependent variable and the independent variables are contained, the exact nature of the dependency is hidden in a `DependencyCharacterization` for now. The `CallpathModel` can be extracted by recursively traversing the system, starting at all possible entry points and iterating over all `Calls` in the entry points. Then for each resulting callpath we can check if any `Dependencies` can be applied. Dependencies that are fully contained within one component do not need to modeled in the `CallpathModel` since the location of its dependents and independents is clear. The same holds true for implicit dependencies i.e. dependencies given by the fact that an external calls parameter are passed to its target.

5.2 Dependency Graph

The `CallpathModel` is a representation of all paths through the system. But it is insufficient as a base for dependency resolution algorithms since it neither shows clearly which variables exist nor does it list all dependencies that could describe these variables. Therefore before resolving the dependencies we transform the `CallpathModel` to a model that contains all information about the system's model variables and dependencies. This is done to simplify the dependency resolution, improve understandability and to provide an extension point to resolve more complex variable and dependency combinations. We chose to use a graph to model the variables and corresponding dependencies with the variables as nodes and the dependencies as edges. Figure 5.3 shows our proposed `RelationshipGraph` model. Each `Node` represents a model variable in combination with a call path leading to this variable. Therefore it refers to a `CallpathPoint` and a `ModelVariable`, as well as the variables distribution. In case the variable's distribution is not yet know, the distribution value is `null`. The dependencies are represented as `Edges`, connecting one dependent `Node` with one or more independent `Nodes`. The `Edge's Equation` describes the equation to calculate the dependents distribution from its independent's distributions, Section 5.4 covers this step. To extract the `DependencyGraph` from the `CallpathModel` we need to iterate over every `CallpathPoint` and create a `Node` for every `ModelVariable` in it and

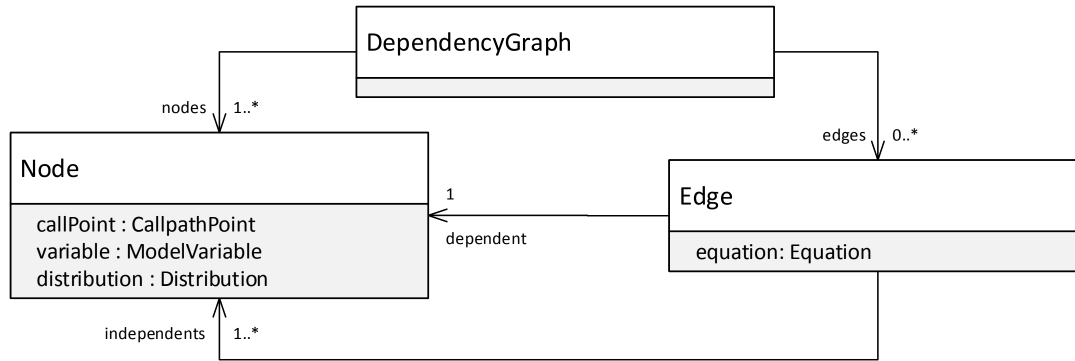


Figure 5.3: DependencyGraph Metamodel

an `Edge` for every dependency defined inside the `CallpathPoint's Behavior`. Then an edge needs to be inserted for every external call parameter and behavior input parameter pair. This means for every two `CallpathPoints` connected by a `CallMapEntry` we need to create an `Edge` connecting the calls parameters with the called behaviors input parameters. Lastly every `Relationship` in the `CallpathModel` needs to be mapped to an `Edge`. This models shows which variables and `CallpathPoint` pairs exist in the model and if a value is known for it. All information about which relationships describe an unknown variable and `CallpathPoint` pair is contained in the `RelationshipGraph`. How these unknown variables can be resolved is described in Section 5.3.

5.3 Relationship Graph Solving

To solve the relationship graph we need to determine the order in which to calculate the edges. This order is relevant because a newly characterized node might allow to resolve additional nodes. It also needs to account for the fact that some dependencies might never be used because an independent node might be impossible to resolve, but the dependent node can also be characterized using another edge. Should it not be possible to resolve values for all nodes the algorithm should still characterize as many nodes as possible. Therefore we propse the algorithm described in 1. The algorithm consists of a loop, that keeps repeating until the underlying dependency graph does not change anymore. Inside the loop it goes over every edge inside the dependency graph and checks if the egde's dependent is not characterized. Should that be the case it checks if all independent nodes of the edge are known and if so we know all input parameters for the edge's `Equation`. This means we can now calculate the value of the edge's independent. Characterizing an edge's dependent is the change that keeps the main loop going. The newly characterized parameter could also be the independent of another edge, therefore we need to check all unsolved edges again. The algorithm terminates if either all parameters are characterized or if some are uncharacterized but can not be resolved using the available data. Therefore a check should be implemented if the value is known when retrieving a value from the solved dependency graph.

5.4 Dependency Calculation

When calculating dependencies there are usually three relevant datatypes, probabilistic logic value, discrete probability distributions and continuous probability distributions. Probabilistic logic values([Nil86]) are boolean values that are uncertain and therefore described using the probability that they are true. If the statement A is true 20 percent

Algorithm 1 Dependency Solver

```
1: function solveDependencies(dependencyGraph)
2:     change = true
3:     while change == true do
4:         change = 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:                     allIndependentsCharacterized = false
10:                end if
11:            end for
12:            if allIndependentsCharacterized == true then
13:                Edge.dependent.value = Edge.equation.apply(Edge.independents)
14:                change = true
15:            end if
16:        end for
17:    end while
18: end function
```

of the time, we would describe it as P(A) = 0.2. Discrete probability distributions are distributions with a fixed number of possible results, like the throw of a dice. These distributions are described using probability mass functions(PMF), which assign a probability to each possible result. Continuous probability functions are distributions with an endless amount of possible results, for example a normal distribution. While they are usually specified using a probability distribution function(PDF) we will instead use Boxed PDFs as described in [Bro14], to simplify the computation of arithmetic operations. For probabilistic logic values only boolean operations can be computed. For the numeric distributions comparisons(LESS, LESSEQUALS, GREATER, GREATEREQUALS and EQUALS) and arithmetic operations(SUM, SUBTRACTION, DIVISION and MULTIPLACTION) need to be calculated. Section 5.4 explains how boolean operations on probabilistic logic values can be computed. Arithmetic operations and comparisons for two discrete distributions, a discrete and a continuous distribution and two continuous distributions are defined in subsections 5.4, 5.4 and 5.4 respectively.

Probabilistic Logic Values

For probabilistic logic values we will compute three operations AND, OR and XOR, which are defined analog to their non-probabilistic counterparts. Koziolek [Koz08] provides the following formulas for these operations:

$$P(C_{AND}) = P(A) * P(B)$$
$$P(C_{OR}) = P(A) + P(B) - P(A \wedge B)$$
$$P(C_{XOR}) = P(A)(1 - P(B)) + (1 - P(A)) * P(B)$$

In these formulas P(A) is short for P(A = true) and C is used to denote the subscript operation being applied to A and B. For the expression A AND B being true both A and be need to be true so we can multiply P(A) and P(B). If either A or B has to be true we can add P(A) and P(B) but this counts the situation where A and B are true twice so we need to subtract P(A AND B) again. If either A or B should be true but not both we

can calculate this as the probability of A being true and B false plus the probability of A being false and B true. All three formulas can be easily computed, since P(A) and P(B) are known.

Discrete Distributions

To apply an arithmetic operation for two discrete distributions we propose to calculate a weighted average over all possible results, with the results probability as weights. The algorithm for the sum of two discrete distributions is shown in algorithm 2. The algorithm

Algorithm 2 Sum of two PMFs

```
1: function calculateSum(pmf1, pmf2)
2:     resultPMF = createEmptyPMF()
3:     for sample1 in pmf1 do
4:         for sample2 in pmf2 do
5:             result = sample1.value + sample2.value
6:             probability = sample1.probability * sample2.probability
7:             updateResultPMF(resultPMF, result, probability)
8:         end for
9:     end for
10: end function
11:
12: function updateResultPMF(pmf, value, probability)
13:     if pmf.contains(value) then
14:         pmf.setProbability(value, probability + pmf.getProbability(value))
15:     else
16:         pmf.put(value, probability)
17:     end if
18: end function
```

first creates a PMF without any entries, then it iterates over all entries of the first PMF. For each entry in the first PMF the algorithm goes over every entry in the second PMF and computes the sum of both values. This value is then added to the newly created PMF using the product of both previous entries as the new probability. When adding a new value to the pmf it first checks if an entry with this value already exists, if so it just addes the new probability to the existing entries probability. Should no entry for this value exist a new entry is added. This algorithm can be reused for other arithmetic operations, by substituting the addition in line five for the desired arithmetic operation. Comparisons between two discrete distributions return a probabilistic logic value, since the result depends on which value each distribution assumes. We used a algorithm similar to algorithm 2, by summing up the probabilities of every possible value combination for which the comparison is true. The resulting algorithm is shown in algorithm 3 for the comparison GREATER. To achieve this the resulting probability is set to zero. Similarly to the algorithm 2 it iterates over every possible combination of entry pairs when taking one entry from each PMF. For these pairs the greater operator is applied and if it returns true the product of the occurance probabilities of each PMF entry is added to the resulting probability.

Discrete and Continuous Distribution

To calculate arithmetic operations and comparisons we reused the algorithms 2 and 3. We iterate over all values of the discrete distribution and all uniform probability distribution

Algorithm 3 GREATER for two PMFs

```
1: function GREATER(pmf1, pmf2)
2:     probability = 0
3:     for sample1 in pmf1 do
4:         for sample2 in pmf2 do
5:             result = sample1.value > sample2.value
6:             if result then
7:                 probability += sample1.probability * sample2.probability
8:             end if
9:         end for
10:    end for
11: end function
```

segments of the BoxedPDF. To adapt the algorithm for a mix of discrete and continuous distributions we need to define the arithmetic operations and comparisons between them. Our prosed formulas for arithmetic operations between discrete values and uniform distribution segments are as following:

$$d + seg(l, u, p) = seg(l + d, u + d, p)$$
$$d - seg(l, u, p) = seg(d - u, d - l, p)$$
$$seg(l, u, p) - d = seg(l - d, u - d, p)$$
$$d * seg(l, u, p) = seg(l * d, u * d, p)$$
$$seg(l, u, p)/d = seg(l/d, u/d, p)$$

In these formulas the discrete value is called d and the uniform probability distribution segment $seg(l, u, p)$ with l as the lower limit, u as the upper limit and p as the probability. When adding or subtracting a discrete value to a uniform probability distribution segment both the segments upper and lower boundries are moved by the discrete value. Multiplying a segment with a discrete value means both borders or the segment are multiplied by the discrete value. This means the segment gets wider if the discrete value is larger than one and smaller otherwise. The total probability stays the same, so the probability density changes. Dividing a segment by a discrete value works analog to the multiplication approach. We were not able to find a solution for the divisio of a discrete value by a uniform probability distribution segment. These formulas are only correct if l of both the input segment and the output segment is positive. This is however no issue since we can assume that all performance relevant attributes have positive values. In the unlikely case of negative values extending these formulas would be possible. For comparisons we can assume the probability for $d == seg(l, u, p)$ is zero since there are infinite possible values for $seg(l, u, p)$. Which means if we can compute $d > seg(l, u, p)$ we can derive the results for all other comparisons . For $d > seg(l, u, p)$ there are three possible cases, either d is larger or equal to u, it is less or equal to l or it lies between l and u. In the first two cases the resulting probability is one or zero respectively, and for the third case we can compute the probability that d is larger than $seg(l, u, p)$ as $p * (u - l)/(u - l)$. Figure 5.4 shows a graphical derivation for the aforementioned formula. The area of the large rectangle is the probability p of the uniform distribution segment. Therefore its height which equals its probability density is $p/(u - l)$. The gray area represents the probability of $d > seg(l, u, p)$. Therefore multiplying the height $p/(u - l)$ with its width $d - l$ results in the probability that $d > seg(l, u, p)$. Using these operations in the algorithms 2 and 3 we are able to compute both comparisons and arithmetic operations for a continous value and a continuous distribution

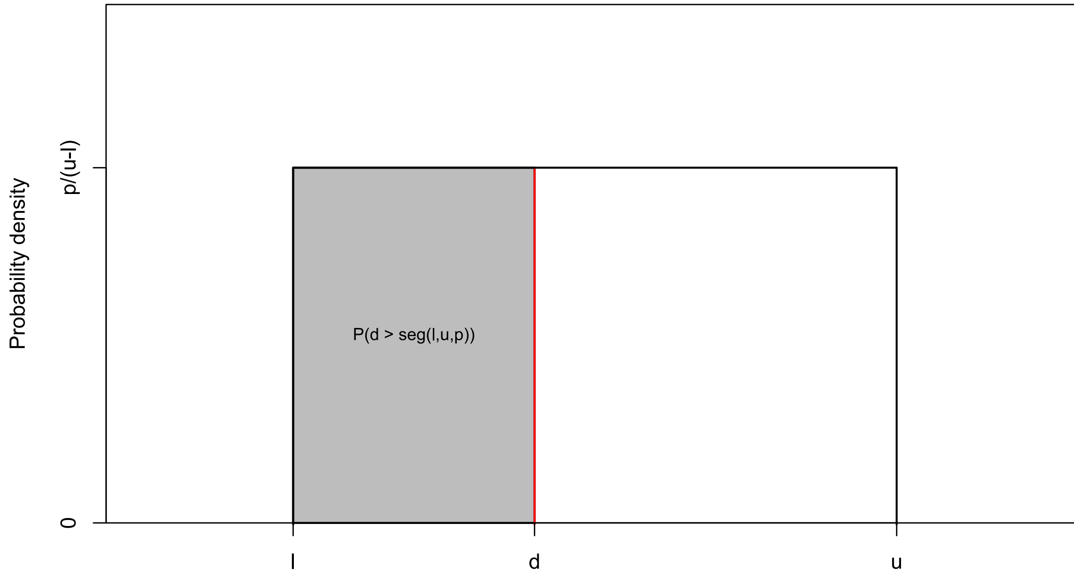


Figure 5.4: Graphical derivation for $d > seg(l, u, p)$

Continuous Distributions

For the calculations in regards to continuous calculations we can again reuse the previously mentioned algorithm 2. As with the discrete/continuous calculations we need to define the operators again. The sum of two continuous distribution is defined as:

$$f(x) = \int_{-\infty}^{\infty} f_1(t) * f_2(t - x)dt$$

We will solve this equation for two uniform probability distribution segments $seg(l_1, u_1, p_1)$ and $seg(l_2, u_2, p_2)$. $f_1(t) * f_2(t - x)$ is zero if $f_1(t)$ is zero, which leads the following equation for $l_1 + l_2 <= x <= u_1 + u_2$:

$$f(x) = \int_{l_1}^{u_1} p_1 * f_2(t - x)dt$$

Meanwhile $f_2(t - x)$ is not zero from $x - t_2$ to $x - l_2$, which means :

$$f(x) = \int_{max(l_1, x - t_2)}^{min(t_1, x - l_2)} p_1 * p_2 dt$$

Which can be simplified to:

$$f(x) = [min(t_1, x - l_2) - max(l_1, x - t_2)] * p_1 * p_2 dt$$

Resolving the min and max to different cases leaves us with:

$$f(x) = \begin{cases} 0 & x < l_1 + l_2 \\ (x - l_1 - l_2) * p_1 * p_2 & l_1 + u_2 \geq x, u_1 + l_2 \geq x \\ (u_1 - l_1) * p_1 * p_2 & u_1 + l_2 < x \leq l_1 + u_2 \\ (u_2 - l_2) * p_1 * p_2 & l_1 + u_2 < x \leq u_1 + l_2 \\ (u_1 + u_2 - x) * p_1 * p_2 & l_1 + u_2 < x, u_1 + l_2 < x \\ 0 & x > u_1 + u_2 \end{cases}$$

This means we have two uniform segments and a linear rising as well as a linear falling section. Upon closer inspection we can see that in no situation both uniform sections will appear. We binned the two linear sections in two equally large bins and used one bin for the uniform section. We do not need the integral to create the bins, since binning is trivial for lines and uniform sections. Should the two bins result in too large of an error we can increase the number of bins used. There is a trade off between execution speed and accuracy here. This cover both addition and subtraction, since for subtractions we can multiply the second segment with minus one and still add them. For multiplications the basic formula is similar:

$$f(x) = \int_{-\infty}^{\infty} f_1(t) * f_2(t/x) * \frac{1}{|x|} dt$$

For $l_1 * l_2 < x \leq u_1 * u_2$:

$$f(x) = \int_{l_1}^{u_1} p_1 * f_2(x/t) * \frac{1}{|x|} dt$$

$f_2(x/t)$ is not zero for x between $min(l_2 * x, u_2 * x)$ and $max(l_2 * x, u_2 * x)$, which leads us to:

$$f(x) = \int_{max(l_1, min(l_2*x, u_2*x))}^{min(u_1, max(l_2*x, u_2*x))} p_1 * p_2 * \frac{1}{|x|} dt$$

We then integrate this equation , since the calculation of bins is not trivial here. With the help of $l_2 < u_2$ we get:

$$F(x) = \begin{cases} 0 & x < l_1 * l_2 \\ \frac{z(1+log(t_1)-log(\frac{x}{t_2}))}{(l_1-t_1)(l_2-t_2)} & t_1 * t_2 < x, l_1 * t2 < x \\ \frac{-z(log(l_1)-log(t_1))}{(l_1-t_1)(l_2-t_2)} & t_1 * t_2 < x, l_1 * t2 \geq x \\ \frac{z(log(t_2)-log(l_2))}{(l_1-t_1)(l_2-t_2)} & t_1 * t_2 \geq x, l_1 * t2 < x \\ \frac{z(-1+log(\frac{x}{l_2})-log(l_1)}{(l_1-t_1)(l_2-t_2)} & t_1 * t_2 \geq x, l_1 * t2 \geq x \\ 0 & x > u_1 * u_2 \end{cases}$$

Due to the fragmented nature of the integral, it is not equal to the cdf. It can only be used to calculate probabilities within one case, to calculate the probability of an area covering multiple cases it is necessary to split it into multiple parts residing in only one case and add up the resulting probability. Using this approach we bin the result into ten bins, we chose a higher number of bins because of the erratic nature of the resulting distributions. When comparing two segments, there are three possible cases, either they do not overlap, they have identical limits or they partially overlap. If they do not overlap the result of the comparison is either zero or the product of both probabilities, depending on which is larger. In case of identical limits the result of a comparison is the product of both probabilities divided by two. Figure 5.5 shows an example for two overlapping segments, the first segment from l1 to u1 and the second one from l2 to u2. In a case like this the result is not trivial. Therefore we divide the problem into a subset of simpler problems. We divide the existing segments to create pairs of segments that either do not overlap or have identical borders. In this pairing it is neccesary to pair one segment with a set of segments whos probabilities sum up to the original segments probability.

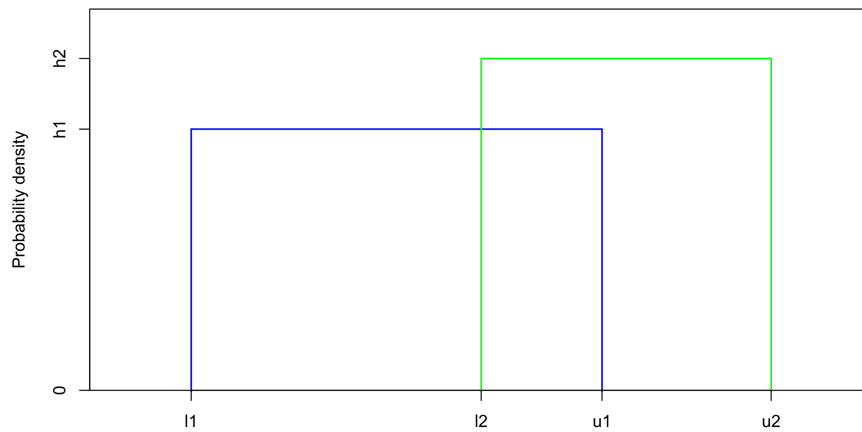


Figure 5.5: Example for overlapping segments

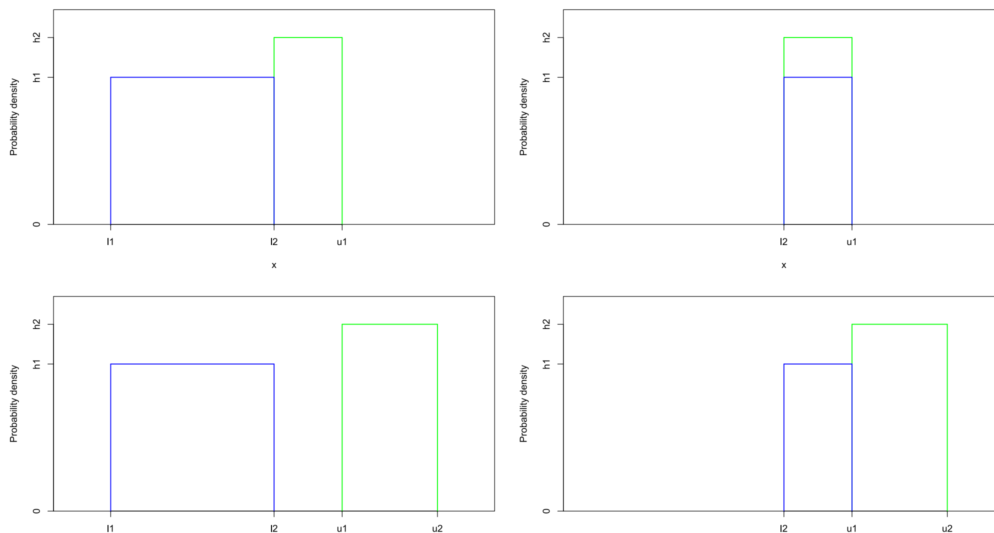


Figure 5.6: Four subproblems created for the comparison of the segments from Figure 5.5

This is shown in Figure 5.6 for the example from Figure 5.5. The solution to the initial comparison is the sum of the results of all subcomparisons. Together with the previously introduced formulas this approach allows us to compute both comparisons and arithmetic operations on two continuous distributions. In total we are therefore able to compute all operations from Section 5.4 between the datatypes we identified as necessary for the calculation of dependencies. An evaluation of the approaches we propose here can be found in Section 8.3. Next we will describe how we implemented this approach for the Descartes Modeling Language (DML).

6. Application of Parametric Dependency Solving to DML

This chapter describes how we apply the approach presented in Section 4.2 to the solving process of DML. We only describe where it is adapted or if we deviate from it. Section 6.1 and Section 6.3 describe how the `CallpathModel` and the `RelationshipGraph` can be used for DML. Their extraction is explained in Section 6.2 and Section 6.4 Since solving the `RelationshipGraph` can be done exactly as described Section 5.4 we will not go into further detail, instead the challenges when calculating values for each dependency are described in Section 6.5.

6.1 Callpath Model

In this section we will explain how we adapted the abstract `CallpathModel` from Section 5.1 to be used with DML. One challenge when adapting the `CallpathModel` for DML is that parameters can not only be contained in the component's behavior descriptions, but also inside the workload description. Therefore we need to use two types of points in the `CallpathModel`. Figure 6.1 shows the resulting meta model.

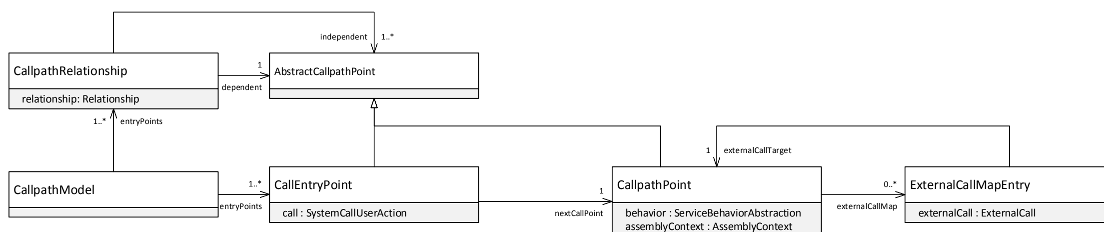


Figure 6.1: DML CallpathModel Metamodel

The `CallpathModel` contains a series of `CallpathEntryPoints`, which each represent a `SystemCallUserAction` from the `UsageProfile`. Each of these `CallpathEntryPoints` points to one `CallpathPoint` which consists of one `ServiceBehaviorAbstraction` and one `AssemblyContext`. The `ServiceBehaviorAbstraction` needs to be contained in the `AssemblyContext's` encapsulated component. Two `CallpathPoints` are equal not only if they have the same `ServiceBehaviorAbstraction` and `AssemblyContext`, they also need to have equal parents (either another `CallpathPoint` or a `CallpathEntryPoint`).

For every `ExternalCall` in a `CallpathPoint`'s behavior it needs to have an `ExternalCallMapEntry`, that links the `ExternalCall` to the `CallpathPoint` that is called from the `ExternalCall`. This means by traversing from a `CallPathEntryPoint` through a series of `CallpathPoints` and `ExternalCallMapEntries` until a `CallpathPoint` has no `ExternalCallMapEntries` we get a possible path a request can take through the system. Since one `ServiceBehaviorAbstraction` can be contained in multiple `CallpathPoints` the `ServiceBehaviorAbstraction's` variables can also be contained in multiple `CallpathPoints`. This means any `Relationships` describing these variables need to also be duplicated. The `CallpathModel` contains `CallpathRelationships` to facilitate this. They each encapsulate a DML `Relationship` and point to the `CallpathPoints` which are used as either dependent or independent in this `Relationship`. They do not contain any information about the actual structure of the relationships, only the coarse-grained information about which `CallpathPoints` participate in the `Relationship`.

6.2 Callpath Model Extraction

The extraction of the `CallpathModel` can be described as a model to model transformation. As input it uses three of the DML submodels, the repository, the assembly and the usage profile. The resource landscape and the deployment are not required since the physical location of a component does not influence its parameterization. Additionally it requires a DML helper model called composition markers model, which denotes for every behavior which granularity of behavior description should be used. In the following we will assume that there is only one behavior description to simplify the notation. The information from these models is transformed to a `CallpathModel` as described in the pseudeo code algorithm 4.

Algorithm 4 Extraction of the Callpath Model

```
1:  function extractCallpathModel()
2:      for systemCallUserAction in usageProfile do
3:          entryPoint = createEntryPoint(systemCallUserAction)
4:          firstPoint = createCallPoint(systemCallUserAction.target)
5:          firstPoint.parent = entryPoint
6:          traverseCallPoint(firstPoint)
7:      end for
8:  end function
9:
10: function traverseCallPoint(callPoint)
11:     for externalCall in callPoint.behavior do
12:         newCallPoint = createCallPoint(externalCall.target)
13:         newCallPoint.parent = callPoint
14:         traverseCallPoint(newCallPoint)
15:     end for
16: end function
```

Algorithm 4 iterates over every `SystemCallUserAction` in the usage profile and in line three creates a `CallEntryPoint` for it. Additionally it creates a `CallpathPoint` in line four for the `SytemCallUserAction`'s target, which is then traversed using the `traverseCallPoint()` function in line six. This function iterates over every `ExternalCall` in the `CallpathPoints` behavior and creates another `CallpathPoint` for the target of every `ExternalCall` as shown in line twelfth. The function recursivly invokes itself no the newly created `CallpathPoint` in line fourteen. Every time a new `CallpathPoint` is created the `CallEntryPoint` or the `CallpathPoint` it is created from is labeled as it's parent, which is

done in lines five and thirteen. The algorithm terminates when a CallpathPoint contains no ExternalCalls.

6.3 Relationship Graph

As descibed in Section 4.2 the CallpathModel is used to extract a RelationshipGraph. First we adapted the RelationshipGraph for DML using the generic concepts of the DependencyGraph described in Section 5.2. Similarly to the CallpathModel the fact that parameters can be contained in both components and the workload descriptions means we need to implement two types of Nodes. Another adaption is the existence of two different edges, one for explicit dependencies and another for the model's implicit dependencies. The resulting meta model is shown in Figure 6.2.

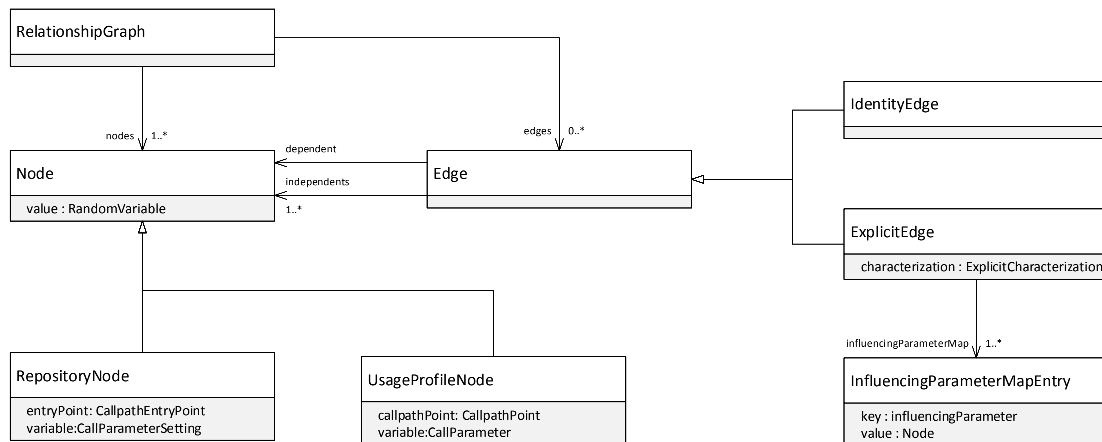


Figure 6.2: DML RelationshipGraph Metamodel

It contains two types of nodes, UsageProfileNodes and RepositoryNodes. Every Usage-ProfileNode represents one SystemCallSetting of a SystemCallUserAction contained in a CallpathEntryPoint, meaning one value that the user passes into the system. A RepositoryNode represents a variable inside the system in combination with a call path, which means that for one variable in the system there can be multiple RepositoryNodes. This is modeled by giving every RepositoryNode a reference to a CallpathPoint and a Callparameter. A Callparameter can be either a ExternalCallParameter, External-CallParameter, ServiceInputParameter, ServiceOutputParameter or an Influenced-VariableReference. Both types of Nodes have a RandomVariable that characterizes the variables value, which can be unset if the value is not yet known. These Nodes are connected by Edges, which each reference on dependendent Node that is characterized by it and one or more independent Nodes which represent the variables used to describe the dependent. An ImplicitEdge describes a relationship which is not explicitly modeled in the DML model, but is implicitly given. For example if component A calls component B, then every ExternalCallParameter of A needs to have the same value as the corresponding ServiceInputParameter in B. There is no characterization needed for these Edges since if we know the independents value we can copy it for the dependents value. DML's explicitly defined relationships are modeled using ExplicitEdges, which contain the Relationship's ExplicitCharacterization. Its InfluencingParameterMapEntrys link the Relationship's influencing parameters with the Node that represents said parameter. The details on how to calculate the results for these ExplicitCharacterizations can be found in Section 6.5.

6.4 Relationship Graph Extraction

The extraction of the `RelationshipGraph` is another model to model transformation that takes a `CallpathModel` as input and transforms it into a `RelationshipGraph` model. It traverses every `AbstractCallpathPoint` and creates a `Node` for every Parameter inside the `AbstractCallpathPoint`'s behavior description as shown in 5. The algorithm iterates

Algorithm 5 Extraction of the Relationship Graph

```
 1: function extractRelationshipGraph(callpathModel)
 2:     for abstractCallpathPoint in callpathModel do
 3:         for parameter in abstractCallpathPoint.behavior do
 4:             createNode(parameter, abstractCallpathPoint.behavior)
 5:         end for
 6:         for relationship in abstractCallpathPoint do
 7:             createExplicitEdge(relationship)
 8:         end for
 9:         checkForImplicitEdges(abstractCallpathPoint)
10:     end for
11:     for relationship in callpathModel do
12:         createExplicitEdge(relationship)
13:     end for
14: end function
15:
16: function checkForImplicitEdges(abstractCallpathPoint)
17:     for parameter in abstractCallpathPoint.behavior do
18:         if containsCorrespondingParameter(abstractCallpathPoint.behavior) then
19:             createImplicitEdge(abstractCallpathPoint, parameter)
20:         end if
21:     end for
22: end function
```

over every `AbstractCallpathPoint` in the `CallpathModel` . For every parameter inside the `AbstractCallpathPoint` a `Node` is created and for every `Relationship` contained in the `AbstractCallpathPoint` a `Edge` is created between the `Nodes` corresponding to the connected parameters. After the `Nodes` for a `AbstractCallpathPoint` are created the algorithm checks if any implicit `Edges` exist between the `AbstractCallpathPoint` and its parent. This is done by checking for every `ServiceInputParameter` and every `ServiceOutputParameter` in the `AbstractCallpathPoint` if the call to it from the parent contains a corresponding characterization. At last for every relationship in the `CallpathModel` an explicit `Edge` is created, this handles `DependencyPropagationRelationships` spanning across multiple components. This algorithm extracts a `RelationshipGraph` model containing `Nodes` for every parameter occurrence and `Edges` for every dependency between them.

6.5 Dependency Calculation

Adapting the dependency calculation for DML is equivalent to calculating the results for `Relationships`. This can be done as described in Section 5.4, with two adaptations. DML allows to nest relationships, meaning one of the arguments of a `TERM` might for example be a `SUM`. This can be dealt with by recursively calculating the relationship bottom-up. Since each operation results in a value that can be used as input for another operation, this guaranties that it is possible to calculate a value for the relationship. The second task

is to map the data types used in DML to the distributions from Section 5.4. An overview for this mapping is shown in Table 6.1. A `BooleanLiteral` can be directly mapped to

DML Datatype	Distribution
BooleanLiteral	Probabilistic Logic Value
IntLiteral	Discrete Distribution
DoubleLiteral	Discrete Distribution
BoxedPDF	Continuous Distribution
NormalDistribution	Continuous Distribution
ExponentialDistribution	Continuous Distribution
DoublePMF	Discrete Distribution
IntPMF	Discrete Distribution
BooleanPMF	Probabilistic Logic Value

Table 6.1: Mapping of DML data types to distributions

a probabilistic logic value A with P(A) = 1 or P(A) = 0 depending on the `BooleanLiterals` value. In a similar fashion we can map a `BooleanProbabilityMassFunction` to a probabilistic logic value, but in this case the probabilities can be read directly from the `BooleanProbabilityMassFunction`. Both `DoubleProbabilityMassFunctions` and `IntProbabilityMassFunctions` are already probability mass functions, so there is no mapping necessary. `IntLiteral` and `DoubleLiteral` can be used as probability mass functions with one value and a probability of one. The mapping for DML's continuous datatypes is a challenge. While we can directly use DML's `BoxedPDFs`, the `NormalDistributions` and `ExponentialDistributions` have to be transformed to `BoxedPDFs`. The probability for a value between μ to $\mu + x\sigma$ in normal distributions can be calculated as following:

$$F(x) = CDF(\mu + x\sigma) - CDF(\mu)$$
$$F(x) = \frac{1}{2}[1 + erf(\frac{x}{\sqrt{2}}) - \frac{1}{2}[1 + erf(0)]]$$
$$F(x) = erf(\frac{x}{\sqrt{2}})$$

We used this formula to create nine bins to approximate the normal distribution, which are shown in Table 6.2. The table shows for every bin the left limit, the right limit and the probability for this bin.

Bin Number	From	To	Probability
1	$\mu - 3\sigma$	$\mu - 2\sigma$	0.023
2	$\mu - 2\sigma$	$\mu - 1.5\sigma$	0.044
3	$\mu - 1.5\sigma$	$\mu - \sigma$	0.092
4	$\mu - \sigma$	$\mu - 0.5\sigma$	0.15
5	$\mu - 0.5\sigma$	$\mu + 0.5\sigma$	0.382
6	$\mu + 0.5\sigma$	$\mu + \sigma$	0.15
7	$\mu + \sigma$	$\mu + 1.5\sigma$	0.092
8	$\mu + 1.5\sigma$	$\mu + 2\sigma$	0.044
9	$\mu + 2\sigma$	$\mu + 3\sigma$	0.023

Table 6.2: Probabilities for binned normal Distribution

An example for a normal distribution with a mean of four and a standard deviation of one can be found in Figure 6.3. The figure shows in blue the density curve of the normal

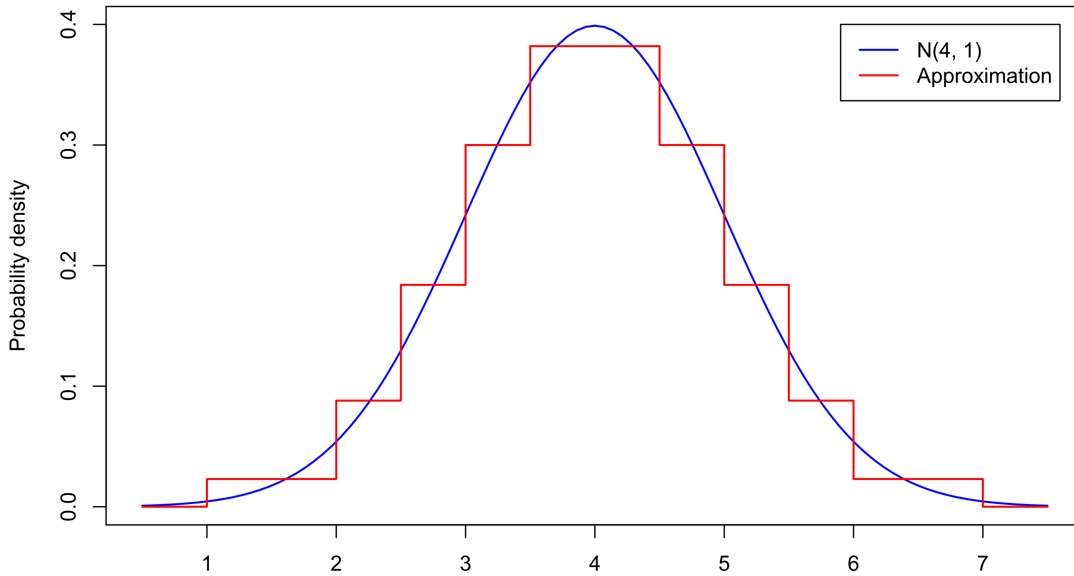


Figure 6.3: Example of the approximation for N(4,1)

distribution and in red our approximation. Since the exponential distribution is a one-sided distribution we will create ten bins, which each have a probability of 9.9%. To calculate the right limit of each bin(the left limit is either zero or the right limit of the previous bin) we resolved the exponential distributions cumulative distribution function for x:

$$CDF(x) = 1 - e^{-\lambda x}$$
$$x = \frac{log(1 - CDF(x))}{-\lambda}$$

The resulting bins are shown in Table 6.3 and Figure 6.4 provides an example for the binning with $\lambda = 0.75$.

Bin Number	From	To	Probability
1	0	$0.10425/\lambda$	0.099
2	$0.10425/\lambda$	$0.22064/\lambda$	0.099
3	$0.22064/\lambda$	$0.35239/\lambda$	0.099
4	$0.35239/\lambda$	$0.50418/\lambda$	0.099
5	$0.50418/\lambda$	$0.68319/\lambda$	0.099
6	$0.68319/\lambda$	$0.90140/\lambda$	0.099
7	$0.90140/\lambda$	$1.18091/\lambda$	0.099
8	$1.18091/\lambda$	$1.57022/\lambda$	0.099
9	$1.57022/\lambda$	$2.21641/\lambda$	0.099
10	$2.21641/\lambda$	$4.60517/\lambda$	0.099

Table 6.3: Probabilities for binned exponential Distribution

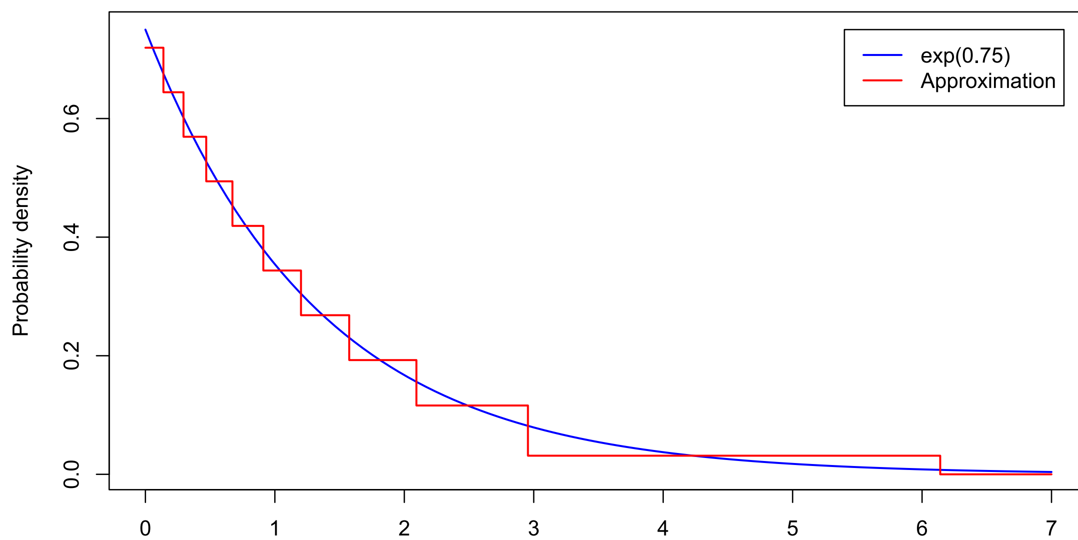


Figure 6.4: Example of the approximation for exp(0.75)

7. Implementation

This chapter will describe the details of our implementation of the dependency resolution approach for DML. First Section 7.1 will list the components we used and describe their functionality. How the controlflow between these components works is shown in Section 7.2. Section 7.3 explains how we integrated this solution in the existing solver. The integration and unit tests used to test the project are explained in Section 7.4 and Section 7.5 contains technical details for our implementation like the languages uses.

7.1 Components

The implementation of our approach to resolve dependencies for `DML` models is separated into multiple components. This allows to reuse parts of it like the dependency calculation and improves maintainability by adhering to the open/closed principle of software design. The implementation is separated in to the following five components:

- **RelationshipSolver** This component wraps the features offered by this master thesis behind an interfaces and acts a general entry point when working with the dependency solver.
- **CallpathExtractor** The extraction of the `CallpathModel` using the transformation from Section 6.2 is managed by this component.
- **RelationshipGrap** The transformation of the `CallpathModel` to a `RelationshipGraph` is contained in this component. For additional information on the extraction of the `RelationshipGraph` see 5
- **RelationshipGraphSolver** The flooding algorithm from Section 5.3 to resolve values for all `Nodes` of the `RelationshipGraph` is managed by this component.
- **RelationshipCaclculator** For a `Relationship` and values for all parameters used in the `Relationship` this component calculates a resulting value as described in Section 5.4 and Section 6.5.

How these components are assembled is shown in Figure 7.1, the `RelationshipSolver` requires the `CallpathExtractor`, the `RelationshipGraphExtractor` and the `RelationshipGraphSolver`. The `RelationshipCalculator` is used by the `RelationshipGraphSolver` to compute a value for a relationship where all parameters are known. How these components interact with each other and can be used to resolve the dependencies insde a DML model is shown in Section 7.2.

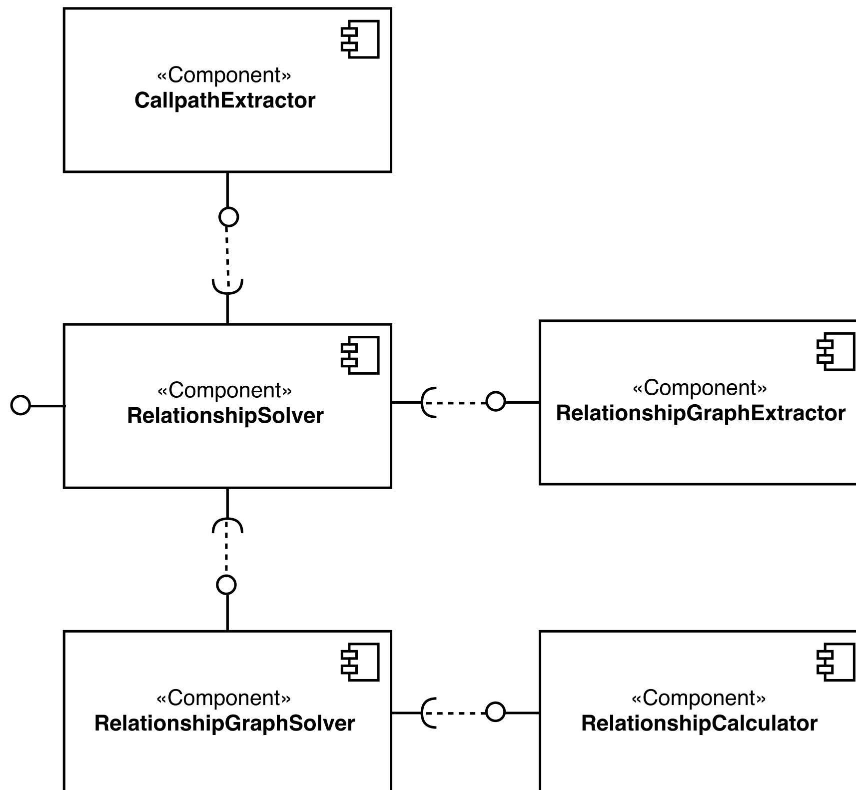


Figure 7.1: Component diagram for the dependency resolution implementation

7.2 Control flow

The components from Section 7.1 work together to resolve the dependencies in a DML model. How such a request is handled is shown in Figure 7.2. As shown in this sequence diagram the `RelationshipSolver` transmits the DML models to the `CallpathExtractor` which returns a `CallpathModel` to the `RelationshipSolver`. This `CallpathModel` is send to the `RelationshipGraphExtractor` which extracts and returns a `RelationshipGraph`. The `RelationshipGraphSolver` is then used to solve this `RelationshipGraph`. To do this it determines the order of the relationship resolution and then uses the `RelationshipCalculator` to determine values for a relationship and a set of input parameters. The returned distribution might be used as an input parameter in another call to the `RelationshipCalculator`. After all relationships are resolved the solved `RelationshipGraph` is send back to the `RelationshipSolver`. This concludes the relationship resolution.

7.3 Integration in existing solver

Prior to this master thesis the process to solve a DQL query on a DML Model was as following. First the model is tailored to the DQL query by selecting the appropriate behavior description abstraction. The `CompositionMarkerModel` details which behavior description abstraction should be used. This information is used to transform the DML Model to a

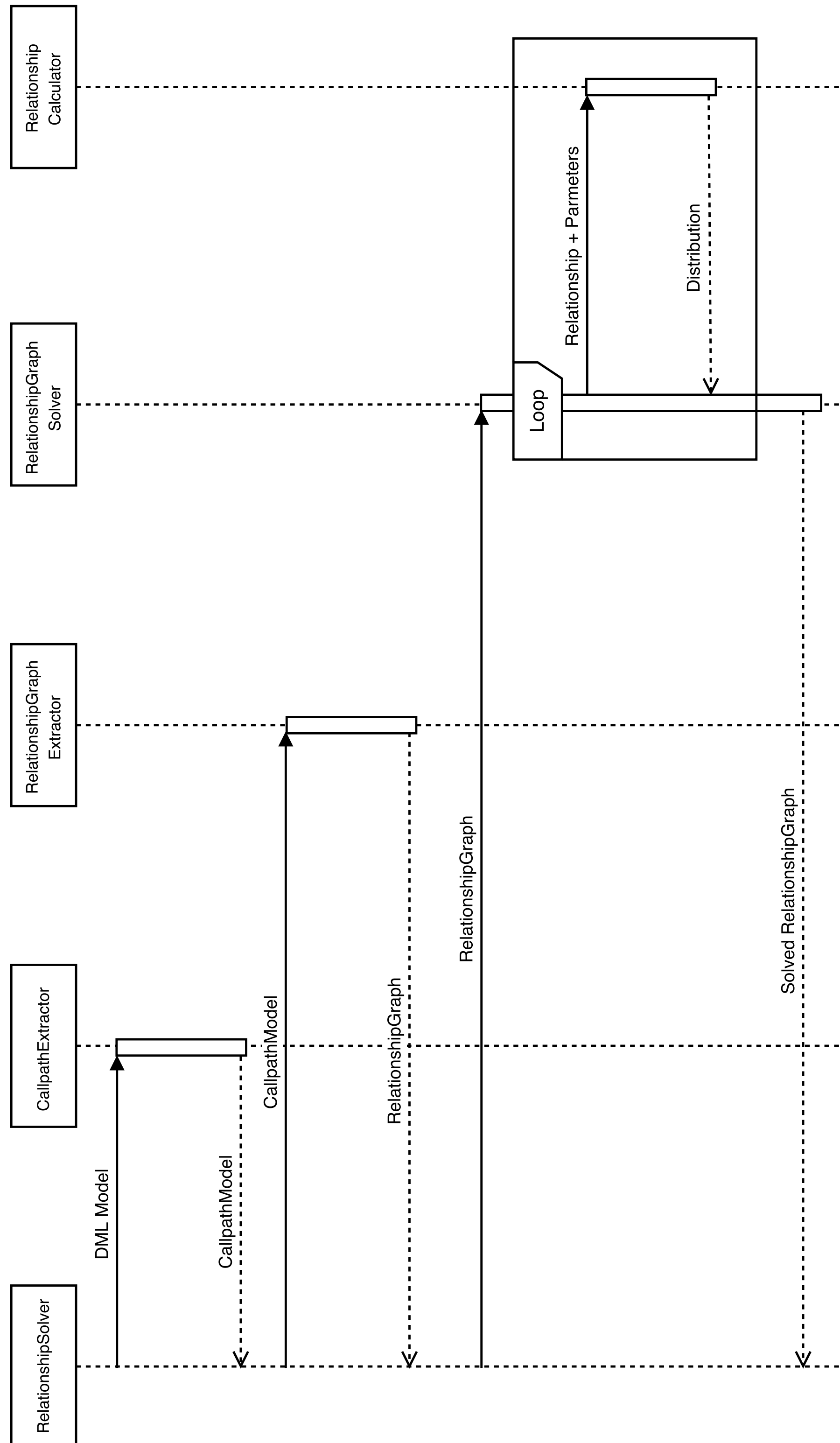


Figure 7.2: Sequence diagram for the dependency resolution implementation

solvable `StackFrameModel`. Next is the tailored model solving, a step in which the DQL query is used to determine the appropriate solution technique. All solution techniques are applied to the `StackFrameModel`. Currently only two solution techniues are supported, a bounds analysis [Jai90] and a transformation to QPN with subsequent simulation using SimQPN. We replaced the current transformation to the `StackFrameModel` with our `RelationshipSolver` component. Since it uses a DML Model and a Composition Marker

Model as input and produces a Stackframe model no iterfaces needed to be changed and the existing tailored model solving can be applied on the output of our `RelationshipSolver`. This allows to automatically solve DML models containing parametric dependencies.

7.4 Testing

Automated testing is a vital piece to ensure a projects health [Pat01]. Unit tests are automated tests that test the functionality of a function, class or code piece. Integration tests on the other hand are end to end tests which do not test specific code but test if a functionality is working as intended. For our project we have five different types of tests: calculation tests, callpath tests, relationshipgraph tests, solving tests and stackframe tests. The first four are unit tests, while the stackframe tests are intended as integration tests.

- **Calculation Test** A calculation test validates if for a set of input distributions and a relationship the expected result distribution is calculated. These expected result distributions are validated as described in Section 8.3.
- **Callpath Test** In a callpath test we evaluate if for a DML model the expected `CallpathModel` is extracted. The expected models are manually created.
- **RelationshipGraph Test** These tests check if the expected `RelationshipGraph` is extracted.
- **Solving Test** Solving tests evaluate if for a `RelationshipGraph` the expected values are resolved.
- **Stackframe Test** Stackframe tests function as end to end tests, which evaluate if our dependency resolution works as intended. They evaluate if for a DML Model the expected `StackframeModel` is returned.

Table 7.1 gives an overview of the number of tests used. Together these tests provide a sufficient test coverage for our implementation.

Tested Feature	# of Tests	Failure Rate
Calculation Test	300	0 %
Callpath Test	12	0 %
RelationshipGraph Test	10	0 %
Solving Test	10	0 %
Stackframe Test	10	0 %

Table 7.1: Number of tests for each type of test.

7.5 Technical Details

Below we list the technical specifications of the project along with some explanations on why they were choosen.

- **Eclipse Plugin** Our implementation is encapsulated in an eclipse plugin. Since the existing DML project is an eclipse plugin as well this allows to easily import the dependency resolution as an dependency, while keeping the builds separatee.
- **Java** The project was done in Java to allow easy wrapping as an eclipse plugin.

- **Java for Transformations** We used Java to implement our transformations instead of a model transformation language like Xtend(`http://www.eclipse.org/xtend/`) or QVTO (`https://projects.eclipse.org/projects/modeling.mmt.qvt-oml`). The main advantages of these languages are a faster execution speed and a syntax tailored to model transformation needs. The execution speed was no issue for us since our Java transformations require only a fraction of a second. While the model transformation syntax is nice we decided that we would rather use a further spread language instead of a niche language since many extension of this project are planned, possibly by students.

- **Eclipse Modeling Framework(EMF)** We used EMF (`https://eclipse.org/modeling/emf/`) to build the `CallpathModel` and the `RelationshipGraph`. This allows for easy reference of model elements from the existing DML model elements since the model from [Bro14] is in EMF. For our automated testing we needed to persist the EMF models. They were saved as XML files using existing EMF functionality.

- **JUnit** Due to the projects Java nature we used JUnit (`http://junit.org/junit4/`) for our unit and integration tests. Alternatively TestNG `http://testng.org/doc/` could be used.

- **Jenkins** We used Jenkins (`https://jenkins.io/`) to automatically build and deploy the project on our build server. This allows for continuous integration and test execution.

8. Evaluation

The evaluation of this master thesis is split into six parts. First we use the Goal Question Metric (GQM) to define what should be evaluated and how it can be measured as shown in Section 8.1. The methodology for our evaluation can be found in Section 8.2. First we evaluate that we calculate the correct result for a given relationship, this is described in Section 8.3. To evaluate if our dependency resolution meets the goals we set in Chapter 4 we build and evaluate a number of example models, which can be found in Section 8.4. Next we evaluated our approach in an end to end scenario first using simple dependencies in Section 8.4 and then using more complex dependencies in Section 8.4.

8.1 Goal Question Metric (GQM)

We choose to use the Goal Question Metric (GQM) approach [Bas92] to give an overview over what we should validate and how we should do it. The GQM approach consists of three steps, first high level goals are defined about what should be accomplished with the evaluation. Next it suggests to collect a number of questions which should be answered by the evaluation. Lastly a number of metrics should be defined by which it can be measured if the questions were answered and therefore if the goals were met. For our dependency resolution approach we identified the following goals:

- Resolve values for all possible model variables
- Support resolution of explicit dependencies
- Integrate approach into existing DML solver

GQM suggests to next find questions based on these goals that depict if the goal was met or not. For our dependency resolution approach we tried to find a question for each goal. If these questions can be answered positively we can assume that our goals have been met. Below are the questions we settled upon:

- Are values resolved for all possible model variables?
- Are the values calculated for explicit dependencies correct?
- Does our integration into the existing DML solver allow to solve models containing explicit dependencies?

The next step is to define metrics which can measure these questions. This was not trivial for our approach, since there are no quantitative metrics to test if a model transformation produces the correct output. In the end we settled on manually building the model we expect as output and comparing it to the algorithms output. Defining a metric for the end-to-end scenario was easier, since we can just compare the predicted response time with measurement values. We came up with the following metrics to validate our approach:

- Similarity of expected and actual output models
- Absolute and relative difference between predicted system response time and expected system response time

How we used these metric to evaluate our dependency resolution is outlined in Section 8.2.

8.2 Methodology

In the chapter we will outline our approach to evaluate this master thesis. Based on the results of Section 8.1 we came up with three steps, the last of which can be split into two separate steps leading to four steps in total. These four tasks are listed below:

- **Evaluate dependency calculation** We will evaluate if our approach calculates the correct result for a relationship and a number of input parameters. It is important here to test for all operations between all datatypes. This is evaluated first, since it can be evaluated independently of the the remaining approach. To evaluate the results of our dependency calculation we compare them to empiric samples.
- **Evaluate dependency resolution for specific modeling features** Next we will evaluate if our approach produces the expected results for models containing specific features. For the selection of these features we used our goals from Chapter 4 as a guideline. We compare the resulting `StackFrameModels` with manually created reference models.
- **Evaluate end-to-end** Lastly we want to evaluate our approach when the whole DML tool chain is used end to end. This means the resulting model is transformed to a QPN and simulated using SimQPN to predict the systems response time. For this modeling a real system and comparing the results with the measurement values proves complicated, since we can not determine why the predicted response time differs from the measurement values. It could be due to a problem with our dependency resolution, deviation cause by discrepancy between the model and the real system or general simulation deviation. Instead we will build a DML equivalent of the PCM MediaStore model and evaluate if we can solve the simple PCM dependencies. Then we will model the same system using the DML specific relationship features like `DependencyPropagationRelationships` and evaluate if we still get the same results.
  - **Evaluation of the MediaStore model** A exact replica of the PCM MediaStore model is created in DML and then we predict the systems response time using both PCM and DML. We assume that both solvers produce similar results for the same model, therefore we can use the results from the PCM solver as ground truth. So any difference in the results can be attributed to a issue with our dependency resolution.
  - **Evaluation of a remodeled MediaStore model** We created a second DML model which depicts the same system as the MediaStore model using DML specific concepts like `DependencyPropagationRelationships`. The results from the PCM model can still be used as ground truth since both models model the same real life system.

8.3 Dependency Calculation Evaluation

Before evaluating our relationship solving we first evaluated if a single dependency is calculated correctly. This means to evaluate if our approach described in Section 5.4 calculates the expected result for a `Relationship` in combination with a set of input parameters. The correctness of these results is both critical for the model to provide correct results as well as hard to uncover that the model provides wrong results due to a problem within these calculations. This leads to the need to rigorously test both the algorithms as well as the implementations. Therefore in addition to the mathematical proofs from Section 5.4 we performed empirical tests to ensure the results correctness. These empirical tests were performed by generating 10.000.000 random numbers from all involved distributions and executing the operations described in the `Relationship` on them.

$$pdf_A(x) = \begin{cases} 1 & \text{for } 3.7 \leqslant x \leqslant 4 \\ 0.249 & \text{for } 4 \leqslant x \leqslant 6.132 \\ 0.196 & \text{for } 6.132 \leqslant x \leqslant 7 \\ 0 & \text{else} \end{cases} \qquad pmf_B(x) = \begin{cases} 0.25 & \text{for } x = -2.2 \\ 0.5 & \text{for } x = 0.9 \\ 0.25 & \text{for } x = 6.4 \\ 0 & \text{else} \end{cases}$$

For example for the addition of the PMF A and the PDF B shown above we created ten million random variables from each distribution and calculated the sum of each pair, resulting in ten million samples from the resulting distribution. We then plotted a small binned histogram for these samples against the distribution calculated by our algorithm. The result of this is shown in Figure 8.1. The black histogram for the frequency density of the samples

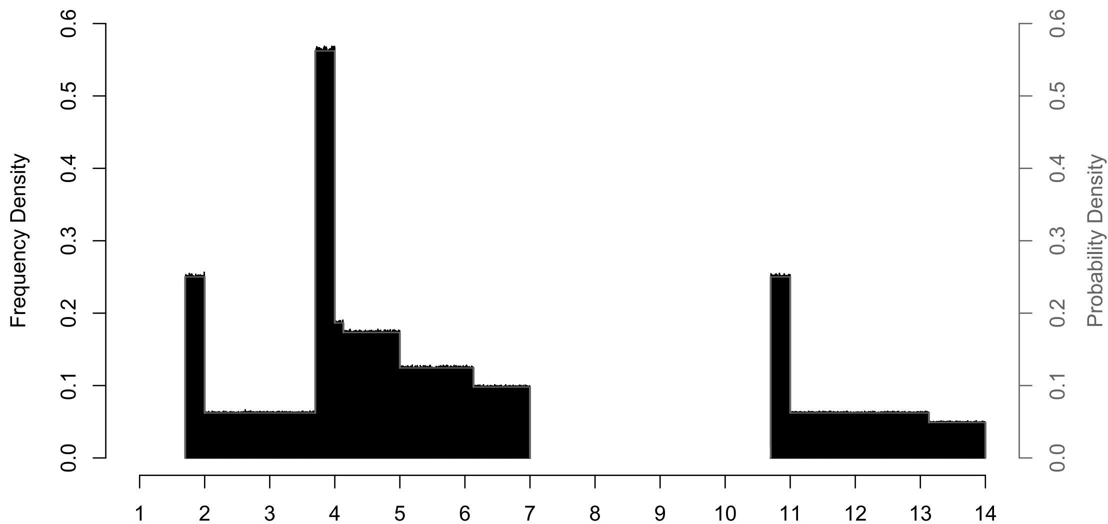


Figure 8.1: Empirical evaluation example for the addition of a `PDF` and a `PMF`

overlaps almost perfectly with the gray probability densities of the calculated distribution, which means our algorithm calculated the correct result. This approach work well for algorithms that provide exact results, but determining if a result is correct is more complicated for the operations between two PDFs, since our restriction to BoxedPdfs causes a loss in precision. An example for this is shown in Figure 8.2. In situations like this it is harder to say if correctly approximated the resulting distribution. We first wanted to use goodness-of-fit tests like the Kolmogorov-Smirnov-Test[Lil67], Anderson–Darling[AD54] test or the Cramér–von Mises test[EL92] but ran into the problem that these tests checked if the samples come from exactly this distribution, which does not work with a binned distribution. The next idea was to compute an existing distance metric for the sample frequencies and the calculated distribution like the Hellinger distance[Ber77], Wasserstein metric[GS+84]

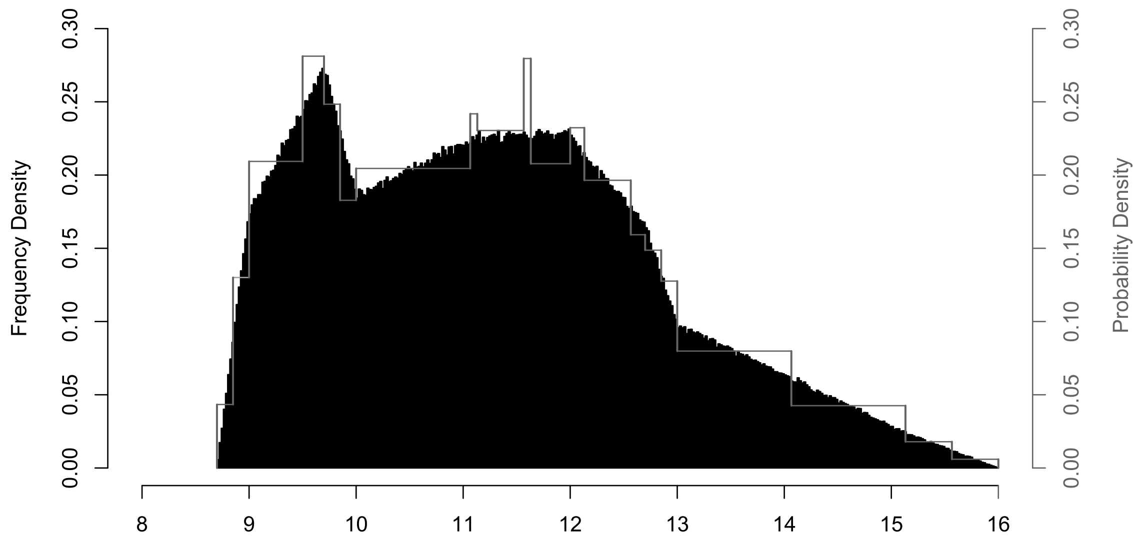


Figure 8.2: Empirical evaluation example for the addition of two `PDFs`

or even the Kolmogorov–Smirnov statistic[Lil67]. But the problem with these metrics is that two distributions having a distance of x has no apparent meaning, and only gets useful when comparing which of two estimations is the better one for example. The main issue is that there are two types of error possible, a systematic error due to mistakes in either the calculation or the computation and an error due to the binning. This second type of error would be acceptable, since that poses a trade-off between computation time required and achieved precision. If the deviations are due to an error of the first type, our dependency calculator would not work properly. To test for this we heavily increased the bin size and therefore eliminated the binning error, this is shown in Figure 8.3 for the addition example from above. From the figure it is obvious that the error was due to the

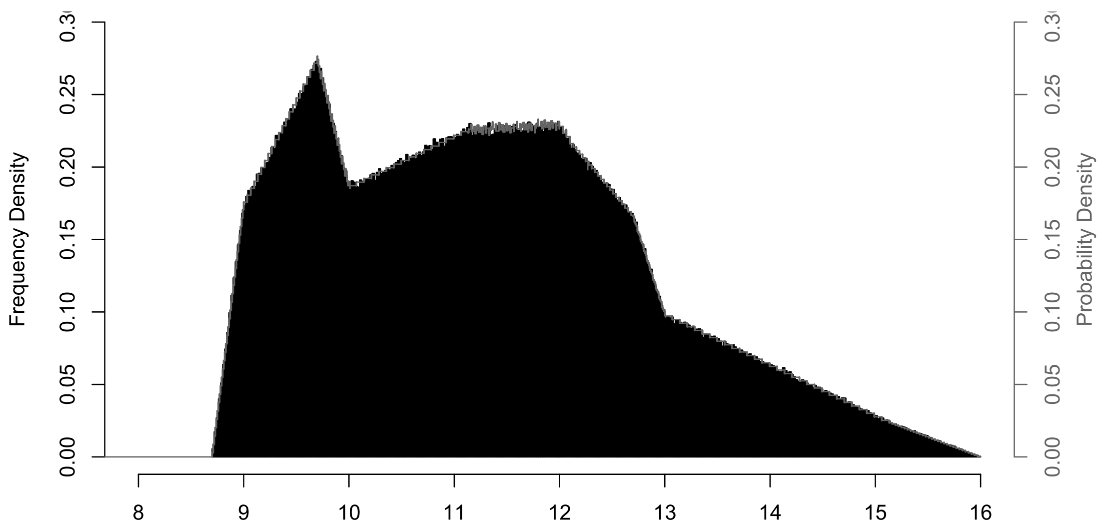


Figure 8.3: Empirical evaluation example for the addition of two `PDFs` with increased number of bins

binning and not a systematic error, we can therefore conclude that our dependency resolution works correctly. Unfortunately this number of bins is not feasible for a production environment, since this heavily influences the computation time. This is especially the case for consecutive operations on PDFs since the resulting number of bins grows exponentially with the number of bins in the two original PDFs. We evaluated a large set of edge cases

for our distribution calculation like this and used these scenarios to write more than 250 junit test cases.

8.4 Dependency Resolution Evaluation

This section will detail how we evaluated our dependency resolution algorithm. We used a set of simple DML models that each contain a small feature that we want to test if our dependency resolution approach can handle it. Each parameter is set as a random decimal value and all operations are sums of the independents and an additional randomly generated decimal number. This will ensures minimal complexity during the dependency calculation, but still prevents any cases where we accidentally get the correct result. For the design of the DML models we used the goals described in Chapter 4 as a guideline. For each model we manually created a reference model what we would consider the correct result of the resultion. Then we compared the results of the dependency resolution with these manually created reference models. We will introduce the test models as traditional UML component diagrams with some additions. The circles inside the components represent the resource demand of a component. The squares represent `ExternalCallParameters`, `ExternalCallReturnParameters`, `ServiceInputParameters` and `ServiceOutputParameters`. A green fill means the value is known while yellow means the value is unknown and has to be resolved using relationships. A thick black arrow between two parameters indicates a `Relationship` between them and a dotted blue arrow represents an implicit relationship. Below follows a list of the models we used. The result of the dependency resolution evaluation is shown in Section 8.4

ExternalCallParameter Model

The first model shown in Figure 8.4 depicts the simplest use case for dependencies in DML, the propagation of an external calls parameter as an input parameter of the target component. Component A defines a `ServiceInputParameter` for its `ExternalCall` to

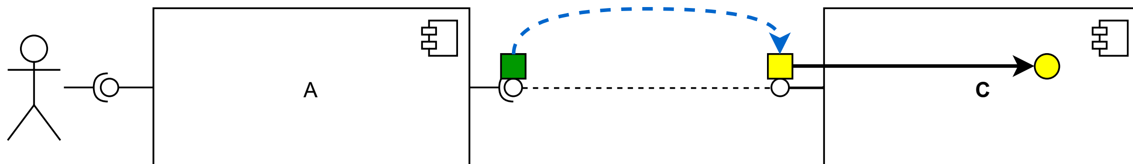


Figure 8.4: ExternalCallParameter Model

component B. This value implicitly characterizes the corresponding `ServiceInputParameter` in C. Then there is a relationship defined between this `ServiceInputParameter` and the `ResourceDemand` in B. This model tests some of the features described in goal 1a and goal 1b.

ExternalCallReturnParameter Model

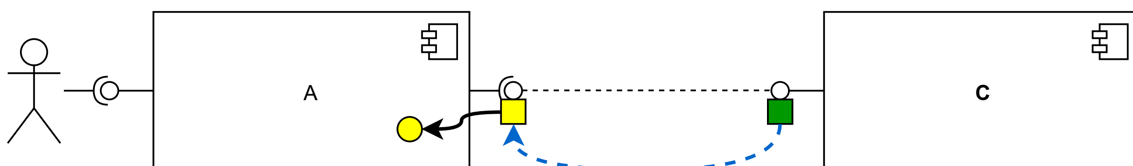


Figure 8.5: ExternalCallReturnParameter Model

This second model is similar to the first one, but as shown in Figure 8.5 it depicts the reverse direction. Here the return parameter of an external call has to be propagated

back to the original component. In this model the resource demand of A depends on the unknown value of a `ExternalCallReturnParameter` which in turn is implicitly dependent on the `ServiceOutputParameter` of component C. This model covers the features of goal 1a and goal 1b that are not yet covered by the first model.

SimpleExteralCallMixedParameter Model

The next model shown in Figure 8.6 is a combination of the first two. A parameter is passed from one component to another and then back.

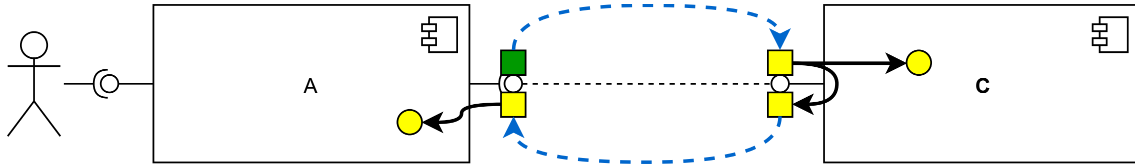


Figure 8.6: SimpleExteralCallMixedParameter Model

In this example component C could be a database, from which component A, a webGUI requests a file via an id (`ExternalCallParameter`) and on the returned file an operation is executed, whose resource demand depends on the file size. Component A passes a parameter to component B, which influences the output parameter of B. This parameter in turn is propagated back to component A and influences a resource demand there. Aside from further testing goals 1a and 1b this example also includes dependency chaining as described in feature 1, since four relationships need to be resolved in order to calculate the resource demand of A.

ComplexExteralCallMixedParameter Model

Figure 8.7 shows a more complex model containing three different components, where the third component can be called in two different ways. Component A serves as an entry point

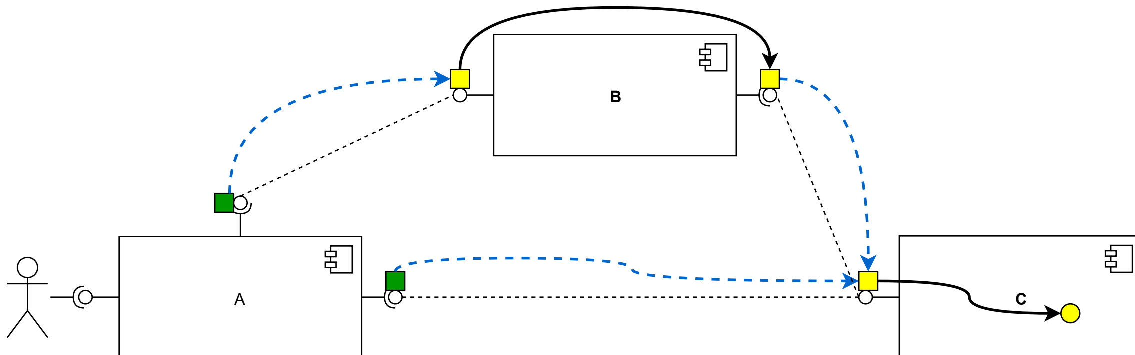


Figure 8.7: ComplexExteralCallMixedParameter Model

for the system, which then calls both components B and C. While the C only contains a resource demand, component B also contains a call to component C. If the call to C comes directly from A it depends on a known `ExternalCallParameter` of A. If A calls B instead a different `ExternalCallParameter` characterizes the value of the `ServiceInputParameter` of B. This `ServiceInputParameter` in turn is the independent in a relationship that characterizes the `ExternalCallParameter` in B, on which the `ServiceInputParameter` of C depends. So depending on which path a request takes the resource demand of C has to be characterized differently. This model tests whether our dependency resolution works for component relationships (goal 1a) and implicit relationships (goal 1b) in models with multiple call paths, as described in 2

SimpleDependencyPropagationRelationship Model

The model shown in Figure 8.8 is the first model containing a `DependencyPropagationRelationship`. It depicts the basic use case for a `DependencyPropagationRelationship` by connecting two parameters in different components directly. This model is similar to

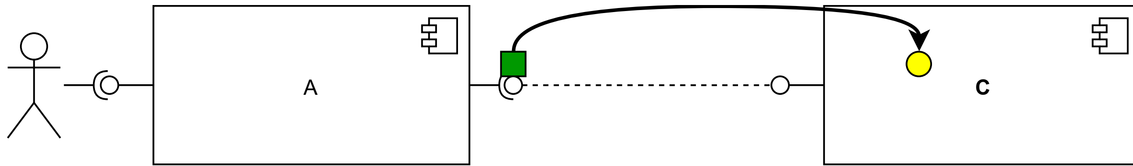


Figure 8.8: SimpleDependencyPropagationRelationship Model

the ExternalCallParameter model, but instead of going from the `ExternalCallParameter` via a `ServiceInputParameter` to the resource demand there is a `DependencyPropagationRelationship` between the `ExternalCallParameter` in a and the resource demand in component C. This model is the first test if goal 1c is met.

ComplexDependencyPropagationRelationship Model

When using `DependencyPropagationRelationships` on of the challenges is the management of multiple call paths. The model in Figure 8.9 depicts such a scenario. This second

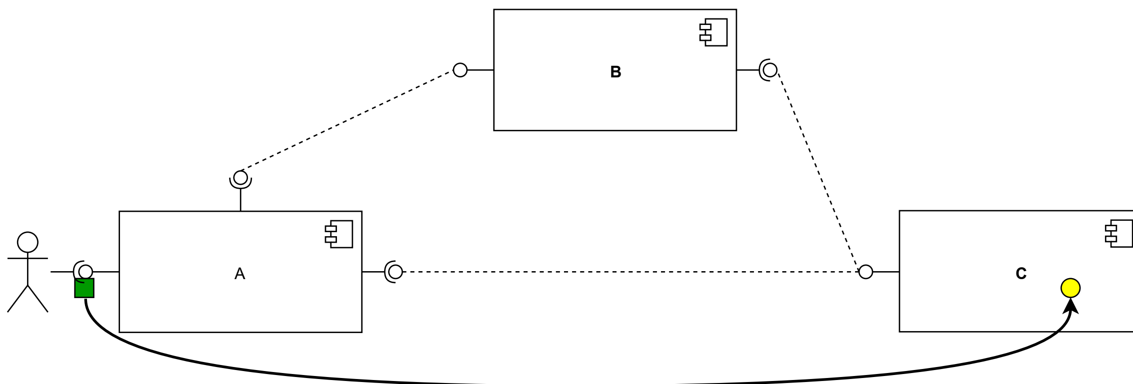


Figure 8.9: ComplexDependencyPropagationRelationship Model

model that tests if goal 1c is met contains three components. As before component A calls to component C and C's resource demand is characterized by a `DependencyPropagationRelationship` with a `ServiceInputParameter` of A as independent. But now component A also calls to a new component B, which in turn call C again. For requests that take this path the resource demand of C should still be characterized by the `DependencyPropagationRelationship` from A, even if the call comes from component C. This model also tests if goal 2 is met when used together with `DependencyPropagationRelationships`.

TwoInstances Model

To validate support for model feature 3 we build a model that describes a scenario in which there are two instances of the component C, as shown in Figure 8.10. The component A calls both components, but with different `ServiceInputParameters`. In each instance of the component C this input parameter influences the resource demand. This means that the resource demands of the two instances need to be calculated separately. This model specifically tests if our dependency resolution works or models containging feature 3.

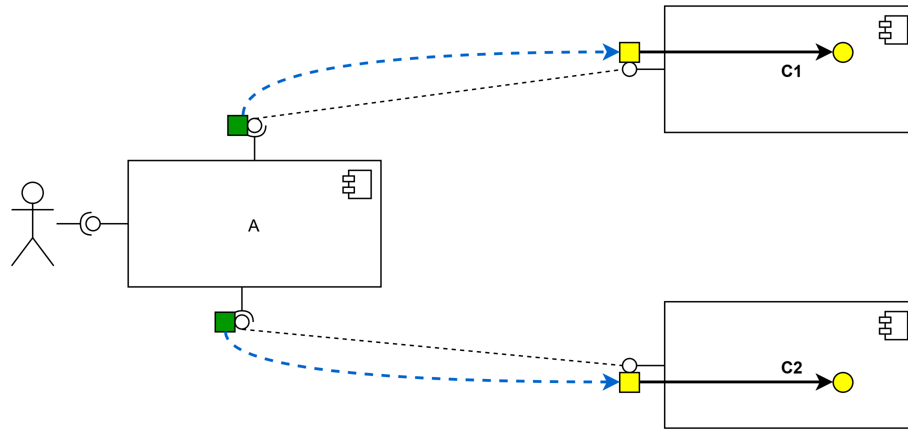


Figure 8.10: TwoInstances Model

TwoCalls Model

Our last model is inspired by our goal to support models with multiple `ServiceBehaviorAbstractions` as described in goal 4. This model can be seen in Figure 8.11. Component A contains two `ServiceBehaviorAbstractions`(not shown in this figure) that

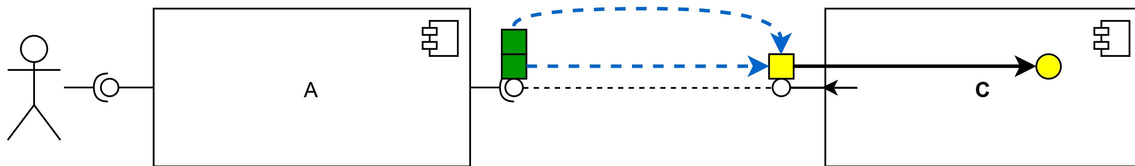


Figure 8.11: TwoCalls Model

both call component C, but with different `ServiceInputParameters`. The resource demand of C should obviously be characterized differently depending on which `ServiceBehaviorAbstraction` the call comes from. If our approach would not be able to resolve this model correctly it would be detrimental, since this is a common scenario.

Results

We used these models to evaluate the dependency resolution by manually creating the expected `StackFrame` model for each model. We then compared these manually created reference models to the `StackFrame` model returned by our implementation of the dependency resolution algorithm. Our test for equivalence tests if all attributes are equal, all

Model	**Figure**	**Attributes**	**Reference**	**Containment**
ExternalCallParameter	8.4	√	√	√
ExternalCallReturnParameter	8.5	√	√	√
SimpleExteralCallMixedParameter	8.6	√	√	√
ComplexExteralCallMixedParameter	8.7	√	√	√
SimpleDependencyPropagation	8.8	√	√	√
ComplexDependencyPropagation	8.9	√	√	√
TwoInstances	8.10	√	√	√
TwoCalls	8.11	√	√	√

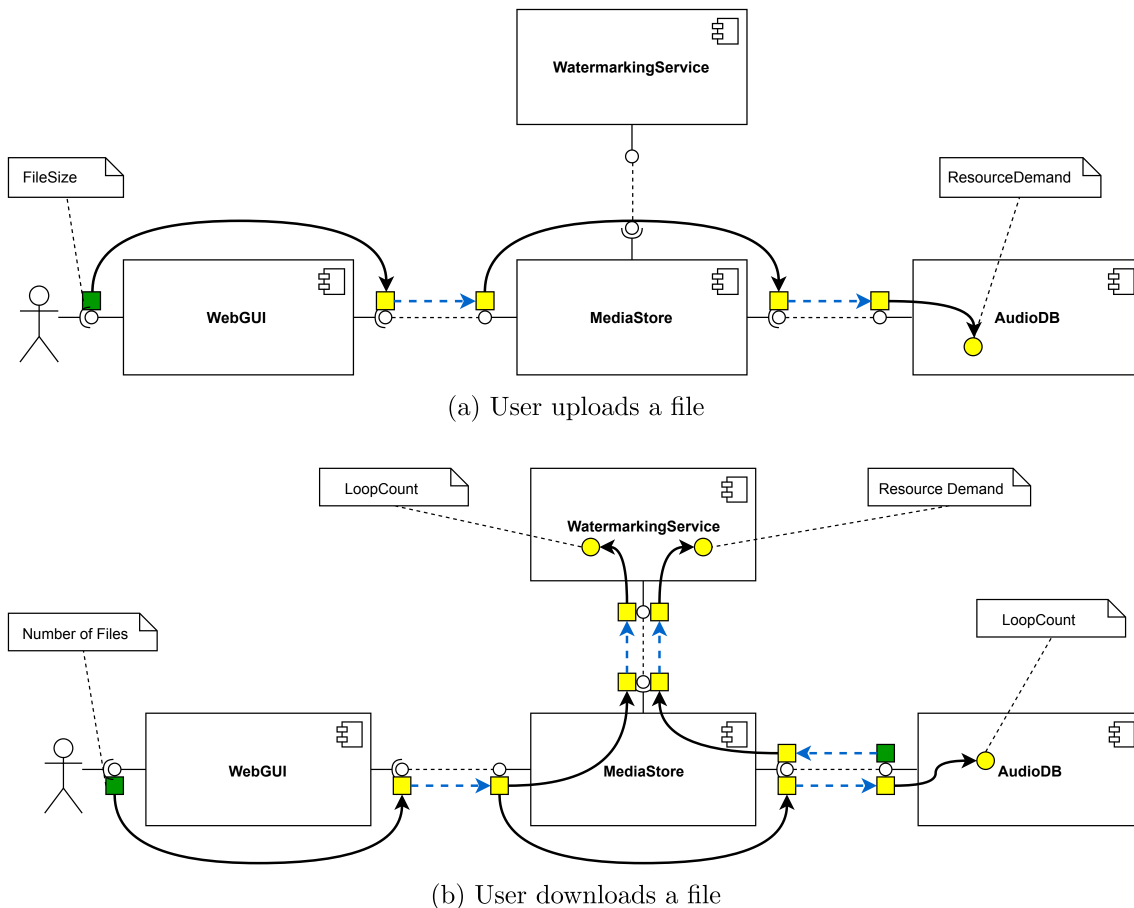
Table 8.1: Equivalence of algorithm output and manually created reference models.

references point to the same elements and in where each model element is contained. As

shown in Table 8.1 the output models of our algorithm are equivalent to the manually created reference models. We additionally created the expected `CallpathModel`, unsolved `RelationshipGraph` and the solved `RelationshipGraph` by hand and used these as well as the `StackFrame` model to create unit tests to both test the implementation and to test if future changes break the dependency resolution.

8.5 Evaluation of the MediaStore model

In addition to evaluating the dependency calculation and the dependency resolution individually, we also want to test our relationship solving in a realistic use case, which means together with the existing QPN solver for DML. So we will test if the calculated resource demands from our relationship solver will lead to accurate performance prediction. The problem with modeling a real world application and comparing the predictions to the actual measurement values is that there might be some systematic error because the model does not capture an aspect of the system. If for example we would have twenty percent deviation between the prediction and the measurement values it would be impossible to say if our dependency resolution worked correctly.The discrepancy might be caused by a mistake in the dependency resolution or be from a systematic error due to the model based approach. We need a scenario where we already know what the correct simulation result would be. For this we build a DML equivalent to the Mediastore model from PCM [BKR09, SK16], an architectural performance model that supports some forms of dependencies.



(a) User uploads a file

(b) User downloads a file

Figure 8.12: UML diagram with parameters and dependencies for the Mediastore

It models a media store similar to apples itunes store. It features two use cases, the upload of new files and the download of existing ones. All user requests are first pointed at a

web gui component, which redirects them to the mediastore component. This component either uploads the new file to the audio database component or downloads the requested one from the audio database. In case of a download the file is sent to a watermarking component before it is returned to the user. The system, its parameters and the dependencies between them are shown in Figure 8.12, the figure uses the same syntax as the figures from Section 8.4. When a user uploads a file, the input parameter filesize is propagated from the WebGUI all the way to the Database where it influences its resource demand. When the user requests a set of files the number of files parameter is propagated to both the AudioDB and the WatermarkingService where it influences a loop count, meaning the number of times a resource demand is executed. The AudioDB additionally returns a file size parameter that influences the watermarking services resource demand. We modeled the system in DML and analyzed it using the proposed dependency propagation algorithm and the existing QPN solver for DML. As shown in Table 8.2 the predicted response time for the system differs by less than two percent from the response time PCM predicts, the remaining difference can be attributed to differences in the solvers. The standard devia-

Model	**Mean Response Time**	**Std. Dev. Response Time**
PCM MediaStore	1.345916	0.9081705
DML MediaStore	1.321071	0.8841287
Relative Deviation	1.778 %	2.719 %

Table 8.2: System response times for the DML and PCM mediastore models

tions differ by less than one percent which is a good indicator that the distributions are similar. To test this hypothesis we plotted a histogram of the response time distributions in Figure 8.13. To further illustrate this we also plotted density curves using kernel density estimation [Sil86] for the resulting response time distributions in Figure 8.14. The fig-

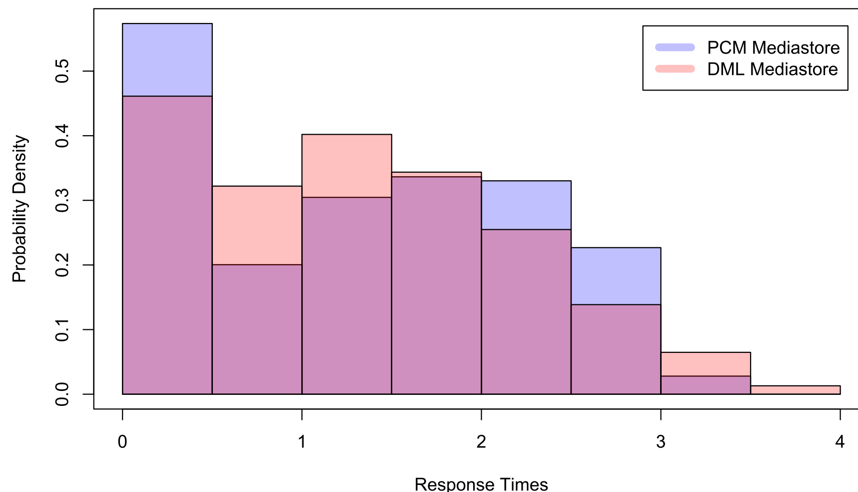


Figure 8.13: Histogram of response time distributions for the DML and PCM Mediastore

ure shows that while the distributions have slight differences they are small enough to be caused by the differences in the two simulation approaches. This means we can conclude that our approach correctly solves models with a dependency complexity similar to PCM. Next we will evaluate if the model still produces correct results if we use the more complex elements of the DML dependency modeling.

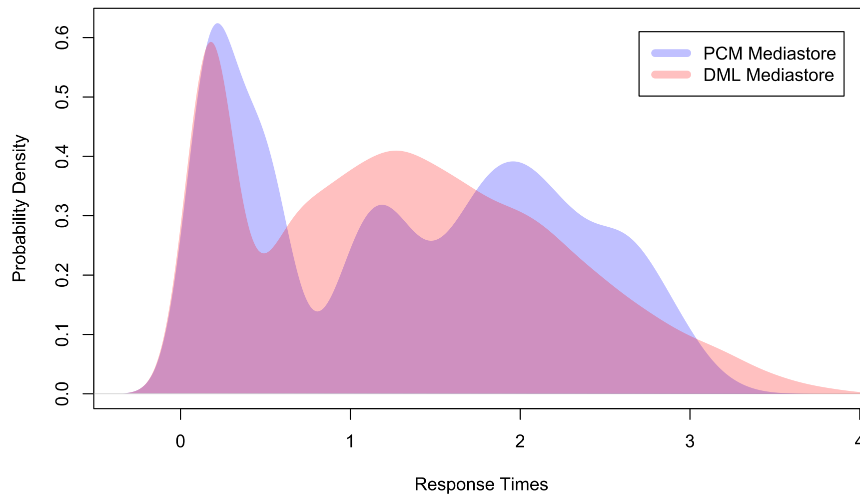


Figure 8.14: Density curves of response time distributions for the DML and PCM Mediastore

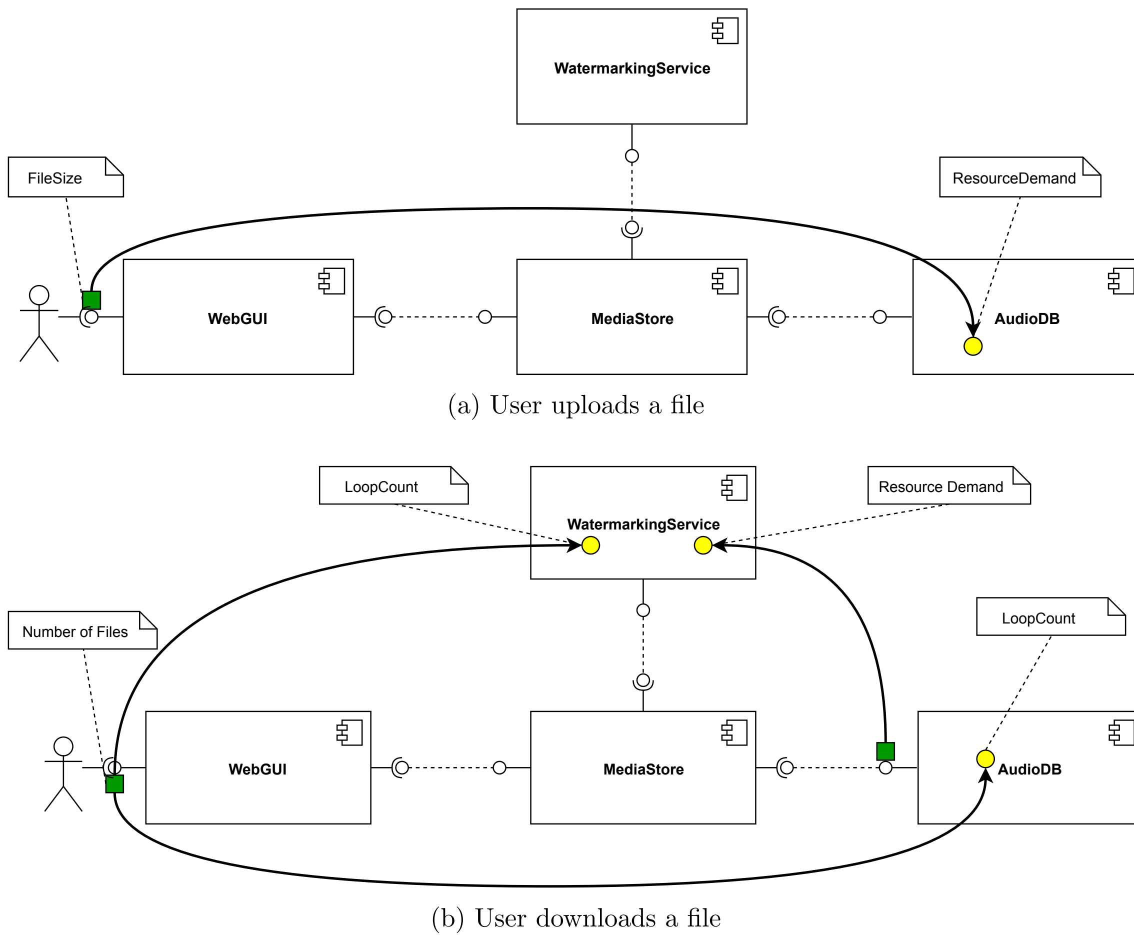
8.6 Evaluation of a remodeled MediaStore model

Now that we know what the expected simulation results are for the media store model, we can evaluate the `DependencyPropagationRelationships` in a end to end scenario. To evaluate this we replaced the component internal dependencies with Descartes Modeling Language (DML) `DependencyPropagationRelationships` which are defined on the assembly level and can therefore span across multiple components. This simplifies the modeling process and make the model easier to understand at the cost of reducing the reusability of individual components. The resulting model is shown in Figure 8.15. This model is equal to the previous model in the sense that they model the same reality, which means that they should return the same results. The DML solver produced a mean of 1.321065 and a standard deviation of 0.883921, as shown in Table 8.3. This shows that the depen-

Model	Mean Response Time	Std. Dev. Response Time
PCM MediaStore	1.345916	0.9081705
DML specific MediaStore	1.321065	0.883921
Relative Deviation	1.321 %	2.7434 %

Table 8.3: System response times for the PCM and DML specific mediastore models

dency resolution returns the correct results for models using `DependencyPropagationRelationships`. We will also evaluate if using the `DependencyPropagationRelationships` provided any benefits. This model used four relationships and three parameters compared to the ten relationships and sixteen parameters contained in the first model. This shows that the use of `DependencyPropagationRelationships` reduced the models complexity, which in turn reduces the effort required to model a system. The `DependencyPropagationRelationships` also provide a more high level view on the model, when comparing Figure 8.12 to Figure 8.15 the second model is a lot more intuitive. For the upload scenario we see that the size of the files the user uploads influences the resource demand of the AudioDB component. In the first model the file size parameter is first passed to the WebGUI, from there to the MediaStore, which in turn passes it as an input parameter to



(a) User uploads a file

(b) User downloads a file

Figure 8.15: UML diagram for the Mediastore using `DependencyPropagationRelationships`

the AudioDB where it influences the resource demand. We can conclude from this case study that our approach can solve both dependencies modeled within components (like PCM) as well as the more complex dependency models of DML.

9. Conclusion

To conclude this thesis a short summary can be found in Section 9.1, followed by a short summary of benefits derived from it in Section 9.2. Lastly, in Section 9.3 presents our ideas for the extension and continuation of the work from this thesis.

9.1 Summary

While performance models have been proven as useful to predict a systems performance aspects, simple performance models have problems with components dynamically changing behavior depending on their assembly. This is caused by the fact that the performance of a component is influenced by different parameters, most prominently the component's input parameters. When changing the component's deployment these parameters can change, which can not be captured by a static component description. A solution to this is to model these parameters and their influence on the component's performance. The Descartes Modeling Language (DML) proposes a sophisticated approach to model these parametric dependencies, but so far no solution algorithm exists. This thesis fills this gap by resolving the dependencies and providing a description of the model parameters that depends on the call's context. First, information about the different paths a request can take through the system is extracted in the `CallpathModel`. This information is then used to create a `RelationshipGraph` in which each node represents a model variable in combination with a call path to it. These nodes are then connected by edges that represent the dependencies between them. Dependencies can either be explicitly modeled or implicit due to the system configurations. The term implicit dependency is used to describe the fact that some variables have to hold equal values even though this fact is not explicitly modeled as a dependency. The input and output parameters of a call to another component have to be equal to this components input and output parameters. Together this forms a directed graph that contains all available information about the model variables and the dependencies between them. On the basis of this graph an algorithm is proposed that uses the already characterized variables and the dependencies between them to derive a characterization for all variables in the system. If not all variables can be characterized, the algorithm still characterizes as many variables as possible. The approach described here was integrated in the existing DML solver by retrieving the corresponding value from the `RelationshipGraph` whenever a model variable is encountered. The setup's evaluation is conducted in three steps. First, the implementation of the dependency calculation is evaluated by comparing the results from the implementation with empirically generated

samples. A set of random numbers from all input distributions is generated and the operations described in the dependency are applied on them. The empirical distribution of this sample is compared to the distribution calculated by the dependency solving algorithm. Secondly, a set of DML models that each contain a specific modeling aspect are created and evaluated if the model is resolved as expected. Lastly, the approach is evaluated for a real world case study. This is done by creating a DML equivalent of a PCM case study model containing simple parametric dependencies. We first evaluated it by comparing the results for both models and after they were identical we used the advanced dependencies of DML to model the same system and evaluated if the results stayed the same. In all three scenarios the approach performed up to expectations.

9.2 Benefit

This section recaps the benefits derived from this master thesis. While the immediate benefit of being able to solve additional features of DML models is obvious some other benefits are less obvious to readers unfamiliar with the vision and ideas behind DML. Below we will list the benefits gained by this master thesis.

- **Solving of additional model features** DML models containing `DependencyRelationships` as well as `DependencyPropagationRelationships` can now be solved. This allows to model and solve parametric dependencies which means that resource demands, response times, loop iteration counts and branch probabilities can be described using an equation that uses other model or workload parameters as input. This equation is then used to calculate context specific values for these variables. Using these modeling features additional real life systems can be modeled and the accuracy of existing models can be improved.

- **Improved model understanding due to intermediate models** Aside from being able to solve additional models the approach also provides two intermediate models. The `CallpathModel` that contains all possible paths through which a request can traverse the system and the `RelationshipGraph` shows all existing model variables and the dependencies between them as a graph. These models help to improve the users understanding of the DML model, since they provide additional views on the model. The `CallpathModel` allows for a different perspective on the assembly of the system, while the `RelationshipGraph` gives detailed information about the models variables and the dependencies between them. This can be especially helpful if due to missing information it is not possible to resolve a value for a variable.

- **Extension point for additional features** The `RelationshipGraph` also serves as an extension point for a large set of additional features, like the characterization of model variable and relationships using monitoring data and a sophisticated decision support between multiple ways to characterize a variable. These features could be implemented by defining additional operations on the `RelationshipGraph`. For more details on this see Section 9.3.

- **Transferability of the approach** In addition to the improvements to DML the approach introduced in Chapter 5 can be applied to other architectural performance models since its description is high level and not DML specific. In theory the approach could also be applied to any data center model and is not restricted to models focusing on performance aspects. Models focusing on reliability or security can directly profit from this approach.

- **Computation of arithmetic for BoxedPDFs** Another, albeit minor contributions is the computation of arithmetic operations for BoxedPDFs. While it is mostly

an application of existing concepts, the availability of a comprehensive overview over the topic can prove useful for other projects dealing with BoxedPDFs.

.

9.3 Future Work

This thesis provides a series of possible extension points that allow for future innovation. There are also some features DML features that are not implemented yet. Ideas for future projects based on this thesis are listed below:

- **Dependency Calculation** In Section 5.4 describes an approach to calculate comparisons and arithmetic operations between different data types. While the current state is sufficient for an evaluation, some steps can be improved and some additional features can be implemented. The binning during the calculation of operations on continuous distributions could be improved in several ways and two of the currently not supported operations could be implemented.
  - **Improved binning** During the computation of arithmetic operations between two continuous distributions there is currently a loss in precision due to the conversion from a PDF to a BoxedPDF. This could be improved by using a better number of bins of not necessarily the same width.
  - **Intelligent composition of multiple resulting PDFs** When calculating a arithmetic operation between two PDFs, currently a set of PDFs is created which is recombined to a single PDF. As the recombination results in exact values, a large number of bins are created. IF multiple of these operations are chained the number of bins increases exponentially, which can be troublesome. This could be avoided by a smarter recombination algorithm.
  - **Improve binning of existing PDF** An alternative solution to the previously mentioned problem of an increasing number of bins is the implementation of an algorithm that reduces the number of bins of an existing PDF. This could be used on the user specified PDFs as well as on the resulting PDF after the execution of an arithmetic operation.
  - **Implement division of a discrete distribution by a continuous distribution** Currently the division of a discrete distribution by a continuous distribution is not supported. While not a frequent event, this could be implemented using approaches like the one introduced in [Cur41].
  - **Implement division of two PDFs** The second unimplemented operation is the divison of two PDFs. A solution might lie in the approach of [Cur41]. Alternatively, an approach that estimates the result with some degree of error could be implemented.
- **Support additional modeling features** DML offers a large collection of parametric dependency features. This thesis did not aim to implement solutions for all features, but was designed with these extensions in mind. DML's `Shadowparameters` could be supported as well as the empirical characterization of both model variables and relationships.
  - **Support Shadowparameter** In some cases the modeler will have knowledge of the existence of a non-functional parameter that influences a components performance that can not be directly mapped to an input or output parameter. To solve this DML introduces the so-called `ShadowParameter` that can be defined inside a component that has no purpose aside from being used in dependencies.

The current approach does not support the usage of these `ShadowParameters`, but could be easily extended to do so by introducing a new type of `Node` in the `RelationshipGraph` model.

- **Characterization of model variables using monitoring data** In DML a variable can be characterized as empirical, meaning it should be characterized at runtime using monitoring data. If the variable is not characterized once, but for each call path individually results will improve. As the `RelationshipGraph` model which lists for every model variable all call paths to it.
- **Characterization of relationships using monitoring data** Relationships can be characterized empirically. This means that measurements for all parameters and variables participating in the relationship have to be available. In this case either an equation can be learned or a machine learning approach can be applied to predict its results.

- **Statistical independence** Solving of the parametric dependencies is based on a simplification. If one variable is used as an independent to describe two or more model variables the distributions for these variables are falsly assumed to be statistically independent. It should be noted that all approaches to resolve parametric dependencies from Section 3.3 make the same simplification. Our `RelationshipGraph` is an excellent tool to check if this issue occurs and which variables are affected. For each variable the variables influencing its characterization are known. This way we can determine for every `Node` a set of `Nodes` that this `Node's` value dependes upon. So if for two variables there is an overlap in these sets of `Node's` we can conclude that they are not statistically independent.
  - **Warning message** A warning message can be displayed to the user containg the information which variables were wrongly assumed to be statistically independent. This solution allows the user to make an informed decision on whether the results are valid.
  - **Generate samples for dependent variables** For solution approaches that allow to specify a list of values that a variable assumes in order instead of a distribution we could generate such lists for the statistically dependent variables. Sets of values from each independent distribution could be generated on which the respective operations can be applied. Unfortunately with increased complexety of the system's control flows difficulties in implementation arise.
  - **Use machine learning to predict values for dependent variables** Another solution would be to ignore the control flow by using measurements for the independent as input in a machine learning algorithm predicting the dependent's values. This approach would still produce incorrect distributions for low level metrics like the response time of a specific component, but should improve results for metrics concerning the whole system.
- **Decision support between multiple ways to characterize a variable** The implementation of empirical characterization of both variables and relationships allows for multiple ways to characterize one variable. These different approaches come with trade-offs concerning precision and time consumption which is important in an online scenario. An automated decision support could be implemented that uses the DQL performance queries [GBK13] specified by the user to tailor the way a variable is characterized to the users concerns.
- **Casestudy** While our evaluation is sufficient for the resolution of the explicit parametric dependencies, the same approach could not be used to implement the mea-

surement based model characterizations. Therefore a case study involving a real system equipped with monitoring software might be necessary. This case study could also be used to prove the accuracy improvements achieved by modeling parameter dependencies.

List of Figures

2.1 Example for parametric dependencies from [Bro14] 10
2.2 Interface meta-model from [Bro14] 11
2.3 InterfaceProvidingRequiringEntity meta-model from [Bro14] 11
2.4 Behavior meta-model from [Bro14] 12
2.5 Assembly meta-model from [Bro14] 12
2.6 Resource Enviroment meta-model from [Bro14] 13
2.7 Resource Specification meta-model from [Bro14] 13
2.8 Deployment meta-model from [Bro14] 14
2.9 Usage profile meta-model from [Bro14] 14
2.10 Variables meta-model from [Bro14] 15
2.11 Parameter meta-model from [Bro14] 15
2.12 Relationship meta-model from [Bro14] 16
2.13 QPN Formalism 16
2.14 Stackframe Metamodel 17
2.15 Mapping for ClosedWorkload [MKK10] 18
2.16 Mapping for ExternalCall [MKK10] 18
2.17 Mapping for Aquire-/ReleaseAction [MKK10] 19

5.1 CallpathModel Metamodel 29
5.2 Example for a callpath model 30
5.3 DependencyGraph Metamodel 31
5.4 Graphical derivation for $d > seg(l, u, p)$ 35
5.5 Example for overlapping segments 37
5.6 Four subproblems created for the comparison of the segments from Figure 5.5 37

6.1 DML CallpathModel Metamodel 39
6.2 DML RelationshipGraph Metamodel 41
6.3 Example of the approximation for N(4,1) 44
6.4 Example of the approximation for exp(0.75) 45

7.1 Component diagram for the dependency resolution implementation 48
7.2 Sequence diagram for the dependency resolution implementation 49

8.1 Empirical evaluation example for the addition of a `PDF` and a `PMF` 55
8.2 Empirical evaluation example for the addition of two `PDFs` 56
8.3 Empirical evaluation example for the addition of two `PDFs` with increased number of bins 56
8.4 ExternalCallParameter Model 57
8.5 ExternalCallReturnParameter Model 57
8.6 SimpleExteralCallMixedParameter Model 58
8.7 ComplexExteralCallMixedParameter Model 58
8.8 SimpleDependencyPropagationRelationship Model 59
8.9 ComplexDependencyPropagationRelationship Model 59

8.10 TwoInstances Model 60
8.11 TwoCalls Model 60
8.12 UML diagram with parameters and dependencies for the Mediastore 61
8.13 Histogram of response time distributions for the DML and PCM Mediastore 62
8.14 Density curves of response time distributions for the DML and PCM Mediastore 63
8.15 UML diagram for the Mediastore using `DependencyPropagationRelationships` 64

List of Tables

6.1 Mapping of DML data types to distributions 43
6.2 Probabilities for binned normal Distribution 43
6.3 Probabilities for binned exponential Distribution 44

7.1 Number of tests for each type of test. 50

8.1 Equivalence of algorithm output and manually created reference models. . . 60
8.2 System response times for the DML and PCM mediastore models 62
8.3 System response times for the PCM and DML specific mediastore models . 63

10. Acronyms

APM Application Monitoring Tool

QPN Queueing Petri Net

SPE Software Performance Engineering

QN Queueing Net

PN Petri Net

UML Unified Modeling Language

PCM Palladio Component Model

DQL Descartes Query Language

DML Descartes Modeling Language

GQM Goal Question Metric

SLA Service Level Agreement

Bibliography

[AD54] T. W. Anderson and D. A. Darling, “A test of goodness of fit,” *Journal of the American statistical association*, vol. 49, no. 268, pp. 765–769, 1954.

[ADEP04] J. Arango, M. Degermark, A. Efrat, and S. Pink, “An efficient flooding algorithm for mobile ad-hoc networks,” in *Proc. of WiOpt*, 2004, pp. 1–7.

[AFG+10] M. Armbrust, A. Fox, R. Griffith, A. D. Joseph, R. Katz, A. Konwinski, G. Lee, D. Patterson, A. Rabkin, I. Stoica *et al.*, “A view of cloud computing,” *Communications of the ACM*, vol. 53, no. 4, pp. 50–58, 2010.

[Akk08] M. Akkouchi, “On the convolution of exponential distributions,” *J. Chungcheong Math. Soc*, vol. 21, no. 4, pp. 501–510, 2008.

[Bas92] V. R. Basili, “Software modeling and measurement: the goal/question/metric paradigm,” Tech. Rep., 1992.

[BdWCM05] E. Bondarev, P. de With, M. Chaudron, and J. Muskens, “Modelling of input-parameter dependency for performance predictions of component-based embedded systems,” in *Software Engineering and Advanced Applications, 2005. 31st EUROMICRO Conference on.* IEEE, 2005, pp. 36–43.

[Ber77] R. Beran, “Minimum hellinger distance estimates for parametric models,” *The Annals of Statistics*, pp. 445–463, 1977.

[BHK11] 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.* IEEE Computer Society, 2011, pp. 183–192.

[BHK12] ——, “Modeling parameter and context dependencies in online architecture-level performance models,” in *Proceedings of the 15th ACM SIGSOFT symposium on Component Based Software Engineering.* ACM, 2012, pp. 3–12.

[BHK17] 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)*, April 2017.

[BK17] 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.

[BKR09] 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, 2009.

[Bre16] P. C. Brebner, “Automatic performance modelling from application performance management (apm) data: An experience report,” in *Proceedings of the*

7th ACM/SPEC on International Conference on Performance Engineering. ACM, 2016, pp. 55–61.

[Bro14] F. M. K. Brosig, "Architecture-level software performance models for online performance prediction," Ph.D. dissertation, Karlsruhe, Karlsruher Institut für Technologie (KIT), Diss., 2014, 2014.

[Cur41] J. Curtiss, "On the distribution of the quotient of two chance variables," *The Annals of Mathematical Statistics*, vol. 12, no. 4, pp. 409–421, 1941.

[EL92] R. Eubank and V. LaRiccia, "Asymptotic comparison of cramer-von mises and nonparametric function estimation techniques for testing goodness-of-fit," *The Annals of Statistics*, pp. 2071–2086, 1992.

[GBK13] F. Gorsler, F. Brosig, and S. Kounev, "Controlling the palladio bench using the descartes query language." in *KPDAYS*, 2013, pp. 109–118.

[GLD04] A. G. Glen, L. M. Leemis, and J. H. Drew, "Computing the distribution of the product of two continuous random variables," *Computational statistics & data analysis*, vol. 44, no. 3, pp. 451–464, 2004.

[GM01] H. Gomaa and D. A. Menascé, "Performance engineering of component-based distributed software systems," in *Performance Engineering.* Springer, 2001, pp. 40–55.

[GMW10] D. Garlan, R. Monroe, and D. Wile, "Acme: An architecture description interchange language," in *CASCON First Decade High Impact Papers.* IBM Corp., 2010, pp. 159–173.

[GS+84] C. R. Givens, R. M. Shortt *et al.*, "A class of wasserstein metrics for probability distributions." *The Michigan Mathematical Journal*, vol. 31, no. 2, pp. 231–240, 1984.

[GS08] S. Gérard and B. Selic, "The uml–marte standardized profile," *IFAC Proceedings Volumes*, vol. 41, no. 2, pp. 6909–6913, 2008.

[HBK12] N. Huber, F. Brosig, and S. Kounev, "Modeling dynamic virtualized resource landscapes," in *Proceedings of the 8th international ACM SIGSOFT conference on Quality of Software Architectures.* ACM, 2012, pp. 81–90.

[HBS+16] N. Huber, F. Brosig, S. Spinner, S. Kounev, and M. Bahr, "Model-based self-aware performance and resource management using the descartes modeling language," *IEEE Transactions on Software Engineering*, 2016.

[HvHK+14] 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.

[Jai90] R. Jain, *The art of computer systems performance analysis: techniques for experimental design, measurement, simulation, and modeling.* John Wiley & Sons, 1990.

[KBH14] S. Kounev, F. Brosig, and N. Huber, "The Descartes Modeling Language," Department of Computer Science, University of Wuerzburg, Tech. Rep., October 2014. [Online]. Available: http://www.descartes-research.net/dml/

[KBHR10] S. Kounev, F. Brosig, N. Huber, and R. Reussner, "Towards self-aware performance and resource management in modern service-oriented systems," in *Services Computing (SCC), 2010 IEEE International Conference on.* IEEE, 2010, pp. 621–624.

[KHB06] H. Koziolek, J. Happe, and S. Becker, "Parameter dependent performance specifications of software components," in *International Conference on the Quality of Software Architectures.* Springer, 2006, pp. 163–179.

[KKR10] 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.

[Koz08] H. Koziolek, "Parameter dependencies for reusable performance specifications of software components," Ph.D. dissertation, Universität Oldenburg, 2008.

[KSM10] S. Kounev, S. Spinner, and P. Meier, "Qpme 2.0-a tool for stochastic modeling and analysis using queueing petri nets," in *From active data management to event-based systems and more.* Springer, 2010, pp. 293–311.

[Law15] A. M. Law, "Statistical analysis of simulation output data: the practical state of the art," in *Proceedings of the 2015 Winter Simulation Conference.* IEEE Press, 2015, pp. 1810–1819.

[LBMAL14] T. Lorido-Botran, J. Miguel-Alonso, and J. A. Lozano, "A review of autoscaling techniques for elastic applications in cloud environments," *Journal of Grid Computing*, vol. 12, no. 4, pp. 559–592, 2014.

[Lil67] H. W. Lilliefors, "On the kolmogorov-smirnov test for normality with mean and variance unknown," *Journal of the American statistical Association*, vol. 62, no. 318, pp. 399–402, 1967.

[MBE13] L. R. Moore, K. Bean, and T. Ellahi, "Transforming reactive auto-scaling into proactive auto-scaling," in *Proceedings of the 3rd International Workshop on Cloud Data and Platforms.* ACM, 2013, pp. 7–12.

[MKK10] P. Meier, S. Kounev, and H. Koziolek, "Automated transformation of palladio component models to queueing petri nets," in *19th IEEE/ACM International Symposium on Modeling, Analysis and Simulation of Computer and Telecommunication Systems (MASCOTS 2011), July*, 2010, pp. 25–27.

[MS84] M. S. Meketon and B. Schmeiser, "Overlapping batch means: Something for nothing?" in *Proceedings of the 16th conference on Winter simulation.* IEEE Press, 1984, pp. 226–230.

[Nil86] N. J. Nilsson, "Probabilistic logic," *Artificial intelligence*, vol. 28, no. 1, pp. 71–87, 1986.

[Pat01] R. Patton, *Software testing.* Sams publishing, 2001.

[Phi] "Robocop website," http://www.extra.research.philips.com/euprojects/robocop/, Phillips. [Online]. Available: http://www.extra.research.philips.com/euprojects/robocop/

[RIF01] M. Ripeanu, A. Iamnitchi, and I. Foster, "Cactus application: Performance predictions in grid environments," in *European Conference on Parallel Processing.* Springer, 2001, pp. 807–816.

[SAM] "Q-impress consortium," www.q-impress.eu/wordpress/wp-content/uploads/2009/05/d21-service_architecture_meta-model.pdf.

[Sch12] L. Schmetterer, *Introduction to mathematical statistics.* Springer Science & Business Media, 2012, vol. 202.

[Sil86] B. W. Silverman, *Density estimation for statistics and data analysis.* CRC press, 1986, vol. 26.

[SK16] M. Strittmatter and A. Kechaou, “The media store 3 case study system,” Faculty of Informatics, Karlsruhe Institute of Technology, Tech. Rep. 2016,1, February 2016. [Online]. Available: http://digbib.ubka.uni-karlsruhe.de/volltexte/documents/3792054

[ST66] M. Springer and W. Thompson, “The distribution of products of independent random variables,” *SIAM Journal on Applied Mathematics*, vol. 14, no. 3, pp. 511–526, 1966.

[ST70] ——, “The distribution of products of beta, gamma and gaussian random variables,” *SIAM Journal on Applied Mathematics*, vol. 18, no. 4, pp. 721–737, 1970.

[VHWH12] A. Van Hoorn, J. Waller, and W. Hasselbring, “Kieker: A framework for application performance monitoring and dynamic software analysis,” in *Proceedings of the 3rd ACM/SPEC International Conference on Performance Engineering.* ACM, 2012, pp. 247–248.

[WBGK15] F. Willnecker, A. Brunnert, W. Gottesheim, and H. Krcmar, “Using dynatrace monitoring data for generating performance models of java ee applications,” in *Proceedings of the 6th ACM/SPEC International Conference on Performance Engineering.* ACM, 2015, pp. 103–104.

[Wel83] P. D. Welch, “The statistical analysis of simulation results,” *The computer performance modeling handbook*, vol. 22, pp. 268–328, 1983.

[WH13] J. Waller and W. Hasselbring, “A benchmark engineering methodology to measure the overhead of application-level monitoring.” CEUR Workshop Proceedings, 2013.

[ZCB10] Q. Zhang, L. Cheng, and R. Boutaba, “Cloud computing: state-of-the-art and research challenges,” *Journal of internet services and applications*, vol. 1, no. 1, pp. 7–18, 2010.

[Zsc10] 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.