Modeling and Simulation of Message-Driven Self-Adaptive Systems

Master Thesis of

Larissa Schmid

at the Department of Informatics
Institute for Program Structures and Data Organization (IPD)

Reviewer:	Prof. Dr.-Ing. Anne Koziolek
Second reviewer:	Prof. Dr. Ralf H. Reussner
Advisor:	Dipl.-Inform. Jörg Henss
Second advisor:	Dipl.-Inf. Martina Rapp

25. November 2019 – 25. May 2020

Karlsruher Institut für Technologie
Fakultät für Informatik
Postfach 6980
76128 Karlsruhe

I declare that I have developed and written the enclosed thesis completely by myself, and have not used sources or means without declaration in the text.

Karlsruhe, 25.05.2020

. . . . . .

(Larissa Schmid)

Abstract

Dynamic systems that are capable of reconfiguring themselves use message queues as a common method to achieve decoupling between senders and receivers. Predicting the quality of systems at the design stage is crucial, as changes within later stages of development have a significantly higher cost attached to them. At the moment, there is no well-established method to represent message queues on an architectural level and predict their impact on the quality of systems. Existing approaches do not model queues explicitly and do not consider queuing effects or details of the messaging infrastructure such as flow control. In order to facilitate a representation thereof, this thesis proposes a meta-model, as well as a simulation interface between a simulation of a component-based architecture description language and a messaging simulation. The meta-model has been realized as an extension to the Palladio Component Model. The interface has been implemented for SimuLizar, which is a Palladio simulator, and a RabbitMQ inspired messaging simulation that adheres to the AMQP 0.9.1 protocol. This enables architectural representation of messaging and predicting quality attributes of message-driven, self-adaptive systems. The evaluation with a case study shows the applicability of the approach and its prediction accuracy for point-to-point communication. Moreover, other quality-related attributes, such as queue length, queue input and output rates, and memory consumption, have been predicted correctly. This provides deeper insights into the quality of a system. We also argue that the approach in this work is capable of simulating self-adaptive message-driven systems that reconfigure based on various metrics.

Zusammenfassung

Dynamische, sich selbst rekonfigurierende Systeme nutzen Nachrichtenwarteschlangen als gängige Methode zum Erreichen von Entkopplung zwischen Sendern und Empfängern. Das Vorhersagen der Qualität von Systemen zur Entwurfszeit ist wesentlich, da Änderungen in späteren Phasen der Entwicklung sehr viel aufwändiger und teurer sind. Momentan gibt es keine Methode, Nachrichtenwarteschlangen auf architekturellem Level darzustellen und deren Qualitätseinfluss auf Systeme vorherzusagen. Existierende Ansätze modellieren Warteschlangen nicht explizit sondern abstrahieren sie. Warteschlangeneffekte sowie Details der Nachrichten-Infrastruktur wie zum Beispiel Flusskontrolle werden nicht beachtet. Diese Arbeit schlägt ein Meta-Modell vor, das eine solche Repräsentation ermöglicht, und eine Simulations-Schnittstelle zwischen einer Simulation einer komponentenbasierten Architekturbeschreibungssprache und einer Nachrichtenaustausch-Simulation. Das Meta-Modell wurde als Erweiterung des Palladio Komponentenmodells realisiert. Die Schnittstelle wurde implementiert für den Palladio-Simulator SimuLizar und eine von RabbitMQ inspirierte Simulation, die dem AMQP 0.9.1 Protokoll folgt. Dies ermöglicht architekturelle Repräsentation von Nachrichtenaustausch und das Vorhersagen von Qualitätsattributen von nachrichtengetriebenen, selbst-adaptiven Systemen. Die Evaluation anhand einer Fallstudie zeigt die Anwendbarkeit des Ansatzes und seine Vorhersagegenauigkeit für Punkt-zu-Punkt-Kommunikation. Außerdem konnten andere qualitätsbezogene Metriken, wie etwa Nachrichtenwarteschlangenlänge, Ein- und Ausgangsraten von Nachrichtenwarteschlangen, sowie Speicherverbrauch korrekt vorhergesagt werden. Das ermöglicht tiefere Einsichten in die Qualität eines Systems. Wir argumentieren weiterhin, dass der Ansatz in dieser Arbeit selbst-adaptive nachrichtengetriebene Systeme, die sich basierend auf verschiedenen Metriken rekonfigurieren, simulieren kann.

Contents

List of Figures

List of Tables

1. Introduction

The rationale for this study is outlined in Section 1.1. Then, the objective is defined in section 1.2.

1.1. Rationale

Nowadays, there is a preference for systems in the context of the Internet of Things or microservice systems to be developed as distributed systems. Hereby, message queues decouple the sending and receiving ends of processing requests, which communicate via a message broker and do not need to explicitly know each other. Asynchronous communication is considered to improve scalability and availability of systems as it facilitates the reduction of waiting times and the application of load balancing by distributing messages to attached consumers. The distribution of messages to consumers can not only be dependent on a fixed scheduling strategy, but also on flow control, which can cause back pressure when consumers do not handle messages as fast as they are sent to them. Elasticity is provided by adding or removing consumers. Such scaling decisions can be taken based on queue metrics [13, 14]. Performance is one of the most important qualities of software nowadays. Not only is it important to provide users with a good user experience, but it also helps to reduce operating costs. However, the performance impact of using message brokers in a system can only be determined at runtime. At the moment, there is no method to represent message queues on an architectural level and predict their impact on the quality of systems. Changes at this stage of the development are significantly more costly than during early stages. Existing approaches do not simulate queuing behavior in detail but abstract it. Also, they do not consider self-adaptive systems. Therefore, an approach for modeling message brokers and queuing behavior within self-adaptive system architectures is needed. Simulating these models will allow for a prediction of their quality attributes.

1.2. Aims and Objectives

The main aim of this thesis has been to predict the quality of message-driven self-adaptive systems. To fulfill the aim, we first propose a meta-model for mapping message-based communication. This meta-model is realized as an extension to the Palladio Component Model. We also provide an approach to formalization of the semantics of the meta-model using Colored Petri Nets. Second, a simulation interface between a simulation of a component-based architecture description and a messaging simulation is presented along with an implementation for the Palladio simulator SimuLizar and an AMQP messaging simulation. We chose SimuLizar because of its capability to simulate self-adaptations. For

deriving quality predictions, the Quality Analysis Lab (QuAL) [25] has been integrated into the messaging simulation as well as the implementation of the simulation interface and extended with messaging-specific metrics and measuring points.

The following research questions are addressed:

RQ1: How can message queues be represented on an architectural level?

RQ2: Can prediction accuracy for message-driven systems be improved using a detailed simulation of the message queues?

RQ3: How can adaptations of the system be mapped to the architectural model and the message queue simulation?

2. Foundations

This chapter covers the foundations of this work. Since this work is about modeling and simulation of message-driven self-adaptive systems, Section 2.1 introduces message-oriented middleware and Section 2.2 self-adaptive systems. Section 2.3 presents model-driven software development as model-driven techniques will be used in this work. We describe the Palladio Approach for modeling and simulating software architectures in Section 2.4. The architectural representation of messaging will be realized as an extension to the Palladio Component Model to enable performance predictions for complete software architectures using messaging as communication paradigm. Section 2.5 introduces Colored Petri Nets. We will use them for a formalization of the semantics of the proposed meta-model. How interoperability between different simulations can be classified is presented in Section 2.6.

2.1. Message-oriented Middleware

Message-oriented middleware (MOM) enables scalability, reliability, and flexibility in distributed systems [6]. Peers can communicate with other peers asynchronously via the sending of messages. Messages sent are stored in queues until the receiving consumer is ready to process them one by one. According to Celar [6], there are two types of MOM: Peer-to-peer messaging, where every peer has a middleware component that is responsible for the discovery of other peers, and broker-based messaging, where every peer only needs to know the broker which is responsible for routing the message. The advantage of broker-based messaging is the loose coupling between sender and receiver [18, p. 322]. In the following, the term MOM will refer to broker-based messaging only.

There are two messaging models, point-to-point communication and publish/subscribe [9]. In point-to-point communication, a message is delivered to one consumer although there can be more consumers to that queue. This makes it a good strategy for load-balancing as well as for the distribution of batch jobs. In contrast, producers publish messages to a specific channel or topic in the publish/subscribe model. Consumers subscribe to these channels or topics if they wish to receive messages from there. Any entity can be either a producer, a consumer, or both. A common application of this model is broadcasting. However, publishers do not have any guarantee on the number of receivers which may even be zero. Publish/subscribe provides high flexibility, whereas the complexity is higher than in the point-to-point model [9]. An example of the application of point-to-point communication are messages about purchases that customers made because they should be processed only once. Price update messages by a headquarter should be delivered to every supermarket the headquarter is responsible for. Hence, publish/subscribe communication can be applied here.

As mentioned above, the point-to-point model provides load-balancing by its property of delivering a message to only one connected consumer. However, flow control can also contribute to load-balancing. When flow control is enabled, producers only get a certain amount of credit and can only send as many messages as they have credit for. When the receiver of a message acknowledges it, they get credit back and can, therefore, send further messages. Flow control can also cause back pressure when producers do not have enough credit to send a message to the desired consumer. They can then either choose to send the message to another consumer they have enough credit for or wait until they get credit back for the originally desired consumer.

Hohpe and Woolfs [18] provide design patterns for building messaging solutions. Patterns important for designing a system using messaging as communication paradigm will be introduced in Section 2.1.1. There are several standards for messaging with multiple implementations each. One standard is the Advanced Message Queuing Protocol (AMQP) which is presented in further detail in Section 2.1.2. RabbitMQ, which supports not only AMQP but also e.g. MQTT and STOMP, is introduced in 2.1.3. The messaging simulation used in this work which is inspired by RabbitMQ will be explained in Section 2.1.4.

2.1.1. Design Patterns for Messaging

One of the main goals of using messaging is to achieve decoupling between senders and receivers. A first approach to that is the use of *Message Channels* where the sender has to know the name of a channel only instead of the name of the receiver. Message channels are unidirectional. To ensure that messages can be read by their receivers, *Datatype Channels* [18, p.111ff.] can be used where all data sent on one channel must have the same type. This means that e.g. price update and purchase information messages are sent via two different channels. Point-to-point and publish/subscribe channels map the respective communication model introduced in Section 2.1 [18, p.101]. Invalid and not deliverable messages can be dealt with by forwarding them to an *Invalid*, respectively *Dead Letter* message channel. This should happen in conjunction with a system monitoring tool that can decide upon further delivery of the message [18, p.101].

Applications can connect to message channels using *Message Endpoints* which *"encapsulate the messaging system from the rest of the application [18, p.96]."* A messaging endpoint can either send or receive messages and is channel-specific.

However, when using channels for decoupling only, the number of channels grows very quickly. Also, an application still has to know where to address its messages itself, i.e. which channels and respective receivers are interested in a message. This could depend on criteria the application is not aware of, e.g. the number of messages passed through the channel so far [18, p.78f.]. *Routers* can help with that by taking responsibility for determining the correct destination of a message. The general functionality of a router includes consuming messages from message channels and republishing them to different channels based on a set of criteria. One criterion could be the properties of the message, environmental conditions, or the state of the router [18, p.80ff.]. A router that only uses the properties of a message to determine its destination is the *Content-Based Router* [18, p.230]. The *Aggregator* is an example of a stateful router: It stores messages until it received a complete set of related messages and then republishes them as a single message composed

of the correlated messages [18, p.269]. Messages are correlated by e.g. their type. If a set is considered complete can be determined by various criteria, e.g. by a field in every message indicating how many messages there are to correlate in total or after a certain amount of time [18, p.272]. When a supermarket decides to extend its product range and requests different suppliers of it for the price they charge for the new product, using an aggregator is a good idea. The aggregator just needs to know how many suppliers were requested and can then wait for the individual replies, correlate the ones regarding the same product, and only forward the reply with the lowest price as soon as all suppliers have answered, which is the completeness criterion in this example.

Though routers provide better decoupling, managing an increasing number of message routers an application can send messages to also becomes cumbersome. Using a central *Message Broker* that receives messages from multiple destinations and routes them provides a solution to that [18, p.323ff.]. The internals of the message broker are implemented using other message routers. However, the message broker has a different scope than a single router because it is an architectural pattern as it concerns the whole organization of a system. Message broker hierarchies help with managing complexity by combining several message brokers, each responsible for only a subnet of the system. If a message should be routed to a destination inside a subnet, the local message broker handles that. If the destination is not inside the same subnet, the message is forwarded to a central message broker which in turn routes it to the relevant subnet. This provides a compromise between having only a single point of maintenance and its complexity.

2.1.2. Advanced Message Queuing Protocol (AMQP)

The Advanced Message Queuing Protocol is an open standard for passing messages between applications. The aim is to enable interoperability between all messaging middleware. It is a connected protocol building on TCP/IP with channels providing the ability to multiplex this connection [15, p.19f.]. Information sent is organized into frames of various types. Though the version 1.0 of AMQP is the most recent, the focus in this work will be on AMQP 0.9.1 as the messaging simulation used, which will be presented in Section 2.1.4, is oriented at RabbitMQ which adheres to AMQP 0.9.1. Not only defines AMQP 0.9.1 a network wire-level protocol but also an Advanced Message Queuing Protocol Model (AMQ model) which *"specifies a modular set of components and standard rules for connecting these [15, p.7]."* We present the AMQ model in Section 2.1.2.1. However, the simulation realizes the Flow Control proposed in the 1.0 version, which will therefore be presented in Section 2.1.2.2 together with a brief overview of the conceptual model of transport of AMQP 1.0.

2.1.2.1. Advanced Message Queuing Protocol Model (AMQ model)

The AMQ model distinguishes between three main types of components, the *exchange*, *message queue* and *binding*. Exchanges receive messages from applications and route them to queues based on the binding of the queue to the exchange. There are *Publisher applications* and *Consumer applications* which the first one publishing, i.e. sending messages

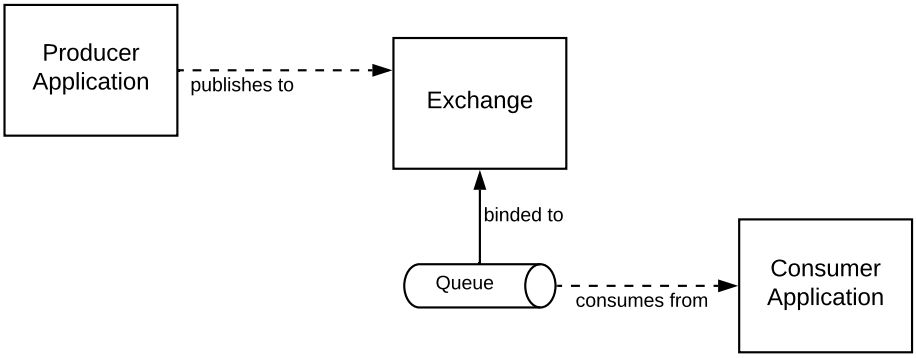

Figure 2.1.: Overview of the entities of the AMQ model

to an exchange and the latter receiving messages from queues. Fig. 2.1 gives an overview of the entities of the AMQ model and their relations.

Message queues are *"named FIFO buffers that hold messages [15, p.28]"*. They can be durable, temporary, or auto-deleted, which refers to their lifetime. Auto-deleted queues are deleted when they are no longer used. Temporary queues are deleted at server shutdown whereas durable queues must be explicitly deleted.

Exchange types are *Direct*, *Fanout*, *Topic*, and *Headers* with the *Direct* and *Topic* exchanges being of particular importance [15, p.15]. In the case of a direct exchange, message queues can bind to it with a routing key K. When a publisher sends a message to that exchange with routing key R, and R = K, the message is forwarded to that queue. Topic exchanges allow for the definition of a routing pattern P instead of a routing key for binding message queues. All messages with a routing key R matching that pattern will then be forwarded to that queue [15, p.27]. Headers exchanges use headers instead of the routing key. Fanout exchanges route incoming messages to all queues regardless of the routing key.

The content of messages consists of a content header and a content body. The content header holds a set of properties, e.g. the routing key, while the content body is binary data to be processed by the receiving application. Messages can also be specified as persistent which means that they are written on disk and can, therefore, be delivered even in the event of malfunction [15, p.26]. Consumer applications can explicitly acknowledge messages, e.g. after receiving or processing them. In the automatic acknowledgment mode, the server removes messages from the queue after delivery [15, p.29]. Consumer applications can use explicit acknowledgments and specify a prefetch to avoid being flooded with more messages than it can process [15, p.29]. The prefetch states how many messages the consumer application accepts before it has to have acknowledged previous messages. To signal publishers that they are over-producing, flow control at the link layer can be applied. Though prefetch and flow control are mentioned in the AMQP 0.9.1 specification, the mechanisms to achieve them are not specified in more detail and are not very sophisticated. Senders can not know how many messages they can send before the receiver blocks; flow control is even described as "emergency procedure" [15, p.29].

2.1.2.2. AMQP 1.0 Transport Model

In contrast to AMQP 0.9.1, AMQP 1.0 focuses more on the actual transport of messages. The conceptual model for transport specifies nodes and links as entities in an AMQP network [28, p.31ff.]. A node can be a Producer, Consumer, or a queue. Nodes are held by containers which are the broker or a client application. Each container can hold many nodes. Client applications and brokers are connected via an AMQP *connection* which is divided into unidirectional *channels*. To enable bidirectional communication between connected entities, AMQP *sessions* correlate channels appropriately [28, p.31]. Messages are transferred via unidirectional *links* [28, p.33] which are established within a session.

Whereas flow control for AMQP 0.9.1 is only available as proprietary extension by different vendors, AMQP 1.0 defines a standardized flow control scheme on sessions [28, p.47] as well as on links [28, p.53]. Flow control on sessions controls the number of transmitted transfer frames that are not acknowledged yet. Message transfers are controlled by flow control on links. For both of them, the receiver can specify a maximum number of events that he accepts without having them acknowledged. The sender can then only send a message if he has received a sufficient number of acknowledgments. When consumers do not send acknowledgments as fast as messages arrive at the queue, back pressure occurs. However, in contrast to the consumer prefetch which is negotiated once at the beginning, the flow control can be dynamically adjusted over run-time.

2.1.3. RabbitMQ

RabbitMQ [30] is an open-source message broker that implements the AMQP 0.9.1 protocol. It also provides several extension to it, e.g. exchange to exchange bindings, dead letter exchanges, and queue length limits. A variety of client libraries, as e.g. Java, .NET, and Ruby, and protocols besides AMQP 0.9.1 are supported. Other protocols are e.g. STOMP, MQTT, and AMQP 1.0 which are supported via plugins.

2.1.4. Messaging Simulation

The messaging simulation [27] we use in this work is inspired by RabbitMQ and adheres to the AMQ model of AMQP 0.9.1. However, on the link layer, it is oriented at AMQP 1.0 which requires conversion of 1.0 frames sent on links to commands from AMQP 0.9.1. In addition to consumers, producers, and queues, which were explicitly described as nodes in the AMQP 1.0 specification, exchanges are also nodes in the simulation. Client applications can be either publisher or consumer applications. Sessions and connections are abstracted in the simulation. By default, flow control is turned on on the link layer. The simulation is still under development. In consequence, it could include bugs and behavior that does not comply to the standard.

The *AmqpBrokerModel* class is responsible for the creation of the entities of the simulation and triggering of sending of messages. It extends the *AbstractSimulationEngine* which is an abstraction layer for discrete-event simulation libraries [1]. The state of discrete systems changes stepwise. Discrete-event simulations realize that by having one central timeline on which events are scheduled, i.e. placed on [33, p.168]. An event

has an event routine that is executed at the scheduled time of the event. Usually, one event schedules subsequent events. For example, the event of receiving a message schedules a disposition event. Simulation entities capable of scheduling events inherit from *AbstractSimEntityDelegator* which is also part of the abstract simulation engine.

2.2. Self-Adaptive Systems

Self-Adaptive systems can configure and reconfigure themselves to react to a changing operational and environmental context. This lets them not only optimize themselves but also enables them to recover themselves and provide additional functionality. Designing systems as self-adaptive systems provides a way to cope with the complexity of software systems by keeping the complexity from users and administrators [7]. Self-adaptive systems can be categorized in proactive and reactive systems [26]: Whereas proactive systems consider anticipated future demands for their reconfigurations, reactive systems reconfigure themselves as a reaction to the current status of the system. A status triggering a reconfiguration could be one or a combination of several performance metrics exceeding or falling below a defined threshold. As it is easier to react to a current state than to predict future demands, the reactive approach has become more popular.

The elasticity of a system refers to its ability to acquire and release resources dynamically. System can be scaled horizontally or vertically. While horizontal scaling adds and releases new machines, vertical scaling affects the sizing, e.g. CPU and RAM, of already existing machines. Because on the most operating systems it is not possible to change their power without rebooting, most cloud providers only offer horizontal scaling [26].

The time a system needs to execute adaptations also has an impact on its effectiveness and efficiency [40]. For example, a newly launched VM is not available immediately; it needs time to boot and be deployed. In fact, an adaptation could also harm the Quality of Service (QoS) of the system when the cost of the reconfiguration in terms of performance is higher than the gained benefit. Therefore, it is important to be aware of the transient effects caused by reconfigurations when developing self-adaptive systems.

2.3. Model-Driven Software Development (MDSD)

Model-Driven Software Development (MDSD) is an approach to software development which treats models as equal to code [39, p.3] and part of the software [39, p.28]. According to Stachowiak [38, p.131ff.], a model has three main characteristics which are *mapping* of an original which can be an object from the real world or another model, *reduction* as not all properties of the original are modeled, and *pragmatism* because the model is designed for a certain context. The goal of MDSD is a reduction of platform complexity [36], a faster software development by automation of activities, and increased platform-independence [39, p.13f.]. Schmidt defines it as a combination of [36]:

- *Domain-specific modeling languages (DSML)* describe the structure and behavior as well as the requirements of an application in a specific domain. The structure of the DSML is defined by a meta-model which will be explained in the following.

- *Transformation engines and generators* transform models to other models or other artifacts, e.g. code, in an automated way. This enables consistency between all artifacts.

Models of the DSML instantiate a meta-model which defines an abstract and concrete syntax and a static and dynamic semantics [39, p.57f.]. The abstract syntax determines which elements and relations between them can be used for modeling independent of their representation in models. How they are represented is specified by the concrete syntax. A meta-model can have several concrete syntaxes but only one abstract syntax. The static semantic defines additional rules and constraints that can not be expressed by the abstract syntax. The semantics of the meta-model is captured by the dynamic semantics.

The Eclipse Modeling Framework (EMF) [12] is a tool that supports MDSD. It defines its own meta-meta-model named Ecore which can then be used to define own meta-models. We will use EMF in this work to define a meta-model for modeling messaging on an architectural level.

2.4. The Palladio Approach

Palladio is an approach for modeling and analyzing component-based software architectures [33, p. 9]. Reussner et al. define software components as in the following: "A software component is a contractually specified building block for software, which can be composed, deployed, and adapted without understanding its internals [33, p. 47]." The Palladio Approach includes three parts which are the Palladio Component Model (PCM), analytical techniques, and a development process model [33, p. 11]. Prediction of performance and reliability is supported. In Section 2.4.1, the PCM is explained in more detail. How measurements during analyses are conducted and stored is presented in 2.4.2. There are multiple tools that aid analysis of systems modeled with Palladio, e.g. SimuCom which is often considered as the reference simulator of Palladio. In this work, we will use SimuLizar for the simulation of models since it supports predicting the quality of self-adaptive systems. Also, SimuLizar is the designated successor and will replace SimuCom soon. Section 2.4.3 introduces SimuLizar.

2.4.1. Palladio Component Model (PCM)

The meta-model of Palladio is structured into viewpoints and view types [33, p.42]. A view that shows elements of a model is always an instance of a view type which defines which classes of the meta-model are shown in that view. Viewpoints group view types that have the same concern. However, one view type can be referenced by several viewpoints.

The Palladio Component Model supports structural, behavioral, and deployment viewpoints [33, p. 44ff.]. The structural viewpoint consists of the repository view type, where components, interfaces, and data types are specified in a system-independent way, and the assembly view type, which contains information about the structure of a system or composite component. Usage models describe the usage of the system whereas Service Effect Specifications (SEFF) specify the intracomponent behaviour and sequence diagrams

View point	System-independent	System-specific
Structural	Repository	Assembly
Behavioral	SEFF	Usage, Sequence Diagrams
Deployment	Resource environment	Allocation

Table 2.1.: Palladio view types

the intercomponent behaviour. These view types make up the behavioral viewpoint. The deployment viewpoint consists of two view types, the allocation and the resource environment view type. While the resource environment view type contains all available containers and their connecting links, the allocation view type describes a concrete allocation of the assembled system to a resource environment. Table 2.1 gives an overview of the viewpoints and view types and whether a view type is system-specific or system-independent. In the following, we will describe the view types relevant to this work in more detail.

2.4.1.1. Repository view type

Repositories contain components, interfaces, and data types. The parameters of signatures of interfaces are typed by data types. Interfaces have a name and contain a list of signatures [33, p.45]. Every signature specifies a name, input parameters, and a return data type. Parameters are also typed by data types. Components specify their relation to other components by defining provided and required roles which refer to interfaces [33, p.47]. To access the services of a component, every component must provide at least one interface. However, it can also require that interface.

In the event extension already present in the PCM, *SourceRoles* and *SinkRoles* inherit from *ProvidedRole*, respectively *RequiredRole*, and specify the ability of a component to send or receive events. *EventGroups* are interfaces specifying *EventTypes* as signatures [31, p.132f.].

There are different abstraction levels components can be specified on to account for different stages in the development process [33, p.51ff.]. For example, in the beginning, it may only be clear which functionality a component should provide but not which functionality it needs in order to do so. Then, a *Provides Component Type* can be specified by only giving provided interfaces of the component. However, if some required interfaces are already known, they can also be given. The *provides component type* is the most abstract component type available. When all required interfaces are known, the specification can be refined by adding them. The component is now specified as *complete component type*. It is also possible to add new provided interfaces. *Implementation component types* are *basic* and *composite* components which additionally specify their behavior. Composite components are implemented by composing other components. How the behavior of components can be specified will be covered in the next section. Fig. 2.2 shows the hierarchy of component types. The *conforms* relationship requires the less abstract component types to provide all interfaces their respective more abstract component type provides and to not require more interfaces than it. However, they may provide more or require fewer interfaces.

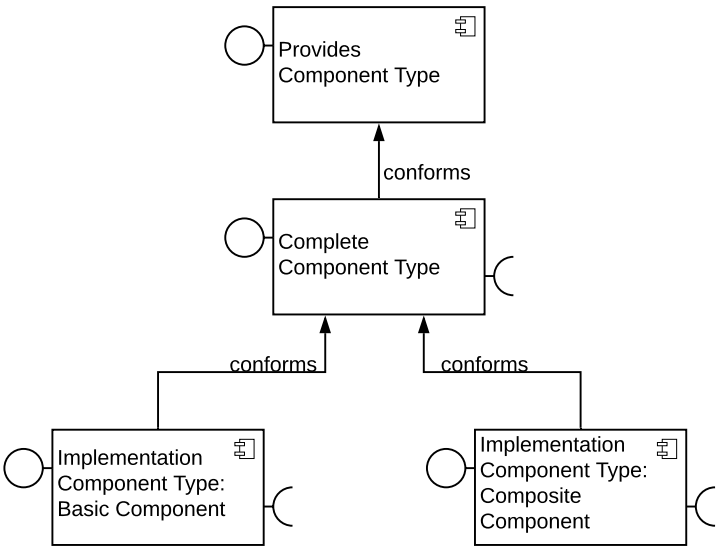

Figure 2.2.: Component type hierarchy

2.4.1.2. SEFF view type

Information about the inner behavior of components is given in SEFFs which specify the relation between provided and required interfaces [33, p.53]. Resource demanding SEFFs (RDSEFFs) are an extension to SEFFs for performance prediction [33, p.99]. They define the sequence of actions performed when a provided service of a component is called. Every action can have a resource demand that specifies how many abstract work units of a certain processing type are needed for performing the action.

In- and output parameters used can be characterized, meaning that their values as well as e.g. bytesize and number of elements in a collection can be given [33, p.106]. Resource demands of actions can depend on the characterization of input parameters. For specifying these characterizations, *Stochastic Expressions (StoEx)* [33, p.103] can be used. This enables stating not only constant values but also random variables with distribution functions. Random variables can be either discrete or continuous; discrete variables are specified with a probability mass function (PMF) and continuous with probability density functions (PDF). Additionally, StoEx allows for calculations and comparisons. The specified behavior in the usage and SEFF models can be dependent on these parameter characterizations, e.g. how long an action takes can be dependent on the bytesize of a parameter.

2.4.1.3. Assembly view type

Systems and composite components are assembled in the assembly view type [33, p.49]. As for components, systems must provide at least one role. Assembly contexts instantiate components from a repository. They inherit the provided and required roles of the encapsulated component. How the assembly contexts of a system or composite component are wired together is given by assembly connectors which link required to provided interfaces

of a subtype or the same type. *AssemblyEventConnectors* can connect source and sink roles. This specifies point-to-point communication between them. Publish/Subscribe communication between source and sink roles is specified by connecting them to an *EventChannel* via an *EventChannelSourceConnector*, respectively *EventChannelSinkConnector* [31, p.138].

Delegation connectors link interfaces of systems or composite components to an interface of an assembly context they contain [33, p.50]. Requests arriving at the interface of the composite structure are then forwarded to the specified assembly context. Delegation connectors can either link provided to provided interfaces or required to required interfaces.

2.4.1.4. Usage view type

For predicting whether a system will conform to a specified service level, usage of the system needs to be taken into account. Palladio offers the possibility to describe user behavior in usage models that can contain several usage scenarios [33, p.56f.]. A scenario describes how a user interacts with the system, i.e. which provided roles of the system he uses in which frequency. Also, specifying the number of users is possible. Therefore, two types of workload are distinguished which are open and closed. While in an open workload, users arrive, execute the scenario behavior, and then leave, users in closed workloads execute the scenario over and over. For open workloads, an interarrival time must be given whereby for closed workloads, the number of expected users and their think time between two executions of their behavior must be specified.

2.4.1.5. Allocation view type

The allocation of a system specifies which entities of the system are allocated to which resource of a resource environment [33, p.58]. Assembly contexts, composite components, and event channels must be allocated by creating *AllocationContexts* that reference the entity to allocate as well as the resource container they should be allocated on. Composite components can be deployed to one resource only.

2.4.2. Quality Analysis Lab (QuAL)

The Quality Analysis Lab (QuAL) is a framework for conducting, storing, and visualizing metric measurements. For conducting measurements, metrics that should be measured need to be specified regarding their capture type (e.g., real number) and unit (e.g., seconds) [25, p.7]. Also, a name and a description must be given to capture the semantics of a metric. An example of a metric is the response time metric which is captured as a real number in seconds. QuAL provides a library of common metrics which can be extended by own metrics. Metrics type the measurements to collect. There are two types of measurements: Basic measurements capture one value while measurement sets consist of several basic measurements [25, p.8].

When running an analysis, probes measure values. Associated calculators process and enrich them with the metric of interest, i.e. converting measures to measurements [25, p.23]. An example of a calculator is the response time calculator which takes two probes

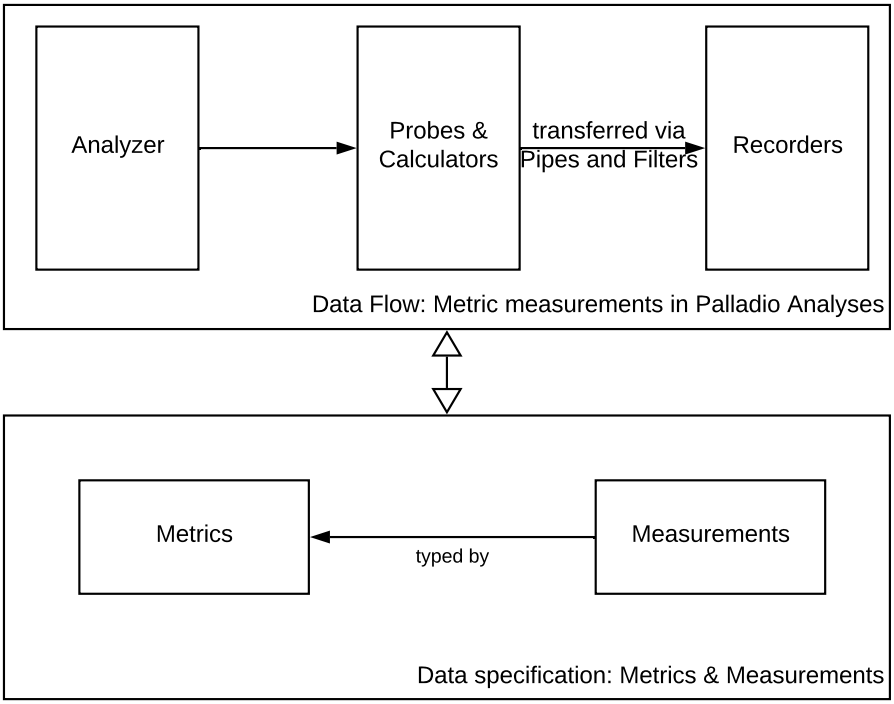

Figure 2.3.: Overview of the parts of the QuAL framework relevant to this work

of the current simulation time and derives the response time by subtracting the two values and attaching the response time metric to the result. Calculators do not only observe probes but are also observed themselves. Whenever a new measurement is available, a recorder gets notified and is responsible for storing it. An example of a recorder is the Experiment Data Persistency & Presentation (EDP2) [25, p.37]. Measurements are transferred to recorders via the Pipes and Filters framework. Fig. 2.3 gives an overview of the mentioned parts of the QuAL framework.

Typically, a set of standard measurements is collected and persisted. However, creating a Measuring Point Repository and Monitor Repository model enables the user to explicitly specify which measurements he wants to take [25, p.39ff.]. At the moment, this is supported by the Palladio analyzer SimuLizar only. The measuring point repository consists of measuring points that specify which part of the system they monitor, e.g. an assembly context. The monitor repository model consists of monitors that each reference a measurement specification, comprising a measuring point, a metric that is to be collected at that measuring point, and if self-adaptations of the system are triggered by these measurements. A monitor can be activated or deactivated.

2.4.3. SimuLizar

SimuLizar extends Palladio with the possibility to model and simulate self-adaptive systems. Therefore, a self-adaptation viewpoint is introduced, containing the Monitoring

Specification View and the State Transition View [4]. The Monitoring Specification View comprises the measuring point and monitor repository model whereas the State Transition View contains self-adaptation rules which consist of a condition and an action to be executed when the condition is true. The system is simulated using a MAPE-K loop [4, 5]: "*The simulated system is monitored, the monitoring results are analyzed, reconfiguration is planned, and a reconfiguration is executed on the simulated system if required.*" Reconfiguration in this context means transforming the PCM model by executing a model-to-model transformation. Supported model transformation languages are QVT-O, Henshin, and Storydiagrams.

On the implementation level, the *SimuLizarRuntimeState* provides access to all simulation and SimuLizar related objects. PCM models are simulated starting from usage scenarios when users perform operations [4]. For simulating users and the system behavior as well as resources, SimuCom and framework code are used. The measurements are conducted as specified in the monitor repository. Whenever a new measurement is available, the Palladio Runtime Measurement Model is updated. This triggers the planning phase which checks if the condition of a self-adaptation rule is true. If that is the case, the respective reconfiguration is executed.

As remarked in Section 2.2, adaptations may even harm the quality of a system if the reconfiguration costs outweigh the gained benefit. Therefore, it is important to be aware of the performance impact of transient effects. Specifying a Self-Adaptation Action Model enables the prediction of transient effects during reconfiguration in SimuLizar [40]. Such a model consists of adaptation actions, each describing one self-adaptation rule. The *BranchingAdaptationStep* maps the condition of the rule. If it is true, the *AdaptationSteps* corresponding to that rule are executed. *EnactAdaptationSteps* call the model transformation and thus map the effect on the configuration of the system. *ResourceDemandingSteps* describe the parametric resource demand of an adaptation.

2.5. Colored Petri Nets

Colored Petri Nets (CP-Nets) are an extension to Petri Nets which are especially useful for modeling concurrent and communicating systems [23, p. 3]. As Petri Nets, they are not limited to a particular domain but provide a general-purpose modeling language. The models are formal in the way that they have a formal definition of syntax and semantic that can be used to prove certain system properties with the stace space method [23, p. 7]. System properties and behavior can be analyzed by executing the model.

CP-Nets is a graphical modeling language [23, p. 14], although the CPN ML programming language supports the definition of data types and annotations describing e.g. which data types a node can handle.

Just as Petri Nets, the model structure consists of nodes, which are places - depicted as ellipses - and transitions - depicted as rectangles -, and directed arcs connecting them. The graphical notation in this work adheres to the notation of Jensen and Kristensen [23]. An arc always connects a place with a transition or vice versa. Even though nodes can have names, they do not have any formal meaning but are there for reasons of clarity. Fig. 2.4 shows an example of a Petri Net with three places and one transition. Place 1 and Place 2

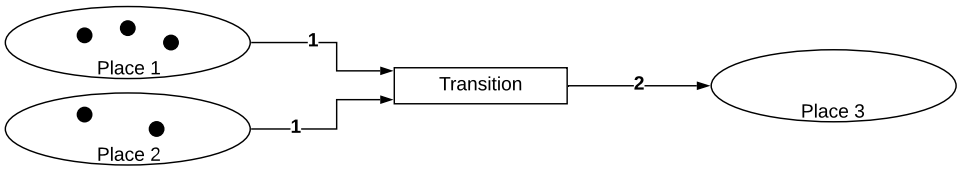

Figure 2.4.: Example of a Petri Net

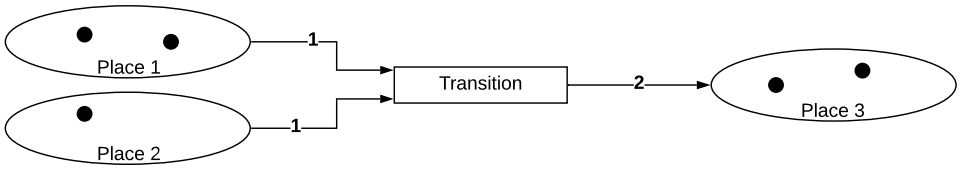

Figure 2.5.: State of the Petri Net after the transition fired once

have initial tokens depicted as black circles inside. When there is one token in Place 1 and one token Place 2, the transition is enabled and fires. The state of the Petri Net after the transition fired once is shown in 2.5. The two tokens required for enabling the transition were transported to Place 3. The transition is still enabled and can fire one more time.

In addition to the elements of Petri Nets, Colored Petri Nets have a concept of data types which is represented by token colors: A colorset containing various token colors defines a data type, e.g. $colorset NUMBERS = int$; defines the data type NUMBERS. Tokens of the data type NUMBERS have an integer as token color [23, p. 15]. It is also possible to define list color sets with the *list* color set constructor [23, p.51]: $colset NUMBERSLIST = list NUMBERS$; defines the data type NUMBERSLIST which comprises a list of tokens of the NUMBERS data type. The ^^ operator concatenates lists. Color sets can also define to be the product of other datatypes, e.g. $colset NUMBERSxNUMBERS = product NUMBERS * NUMBERS$; defines a data type whose tokens are a tuple of $NUMBERS$ tokens. The accepted color sets of each place are by convention annotated next to it, the initial tokens in that place above it. Tokens can be concatenated to multisets using ++ and ′ as operators. The ++ operator takes two multisets and returns their union. How often one element is contained in a multiset can be specified with the infix operator ′ by writing the number of appearances of an element left and the element right to it [23, p.16].

A transition moves tokens in the direction of its arcs. Transitions are enabled when tokens satisfying their arc expression are in their input place [23, p. 17]. If two transitions on a place are enabled, only one can fire; however, if a transition has several outgoing arcs to several output places, the token is added to every output place.

Fig. 2.6 gives an example of a CPN similar to the Petri Net shown before. However, the place one and two now only accept tokens of the color set $NUMBERS$. Place three accepts tokens of the color set $NUMBERSxNUMBERS$, therefore, the two tokens forwarded by the transition are concatenated to a tuple. Definitions of all color sets and variables are given in Listing 2.1. Place one and two both have initial tokens which are given as multisets in

their upper right corner. Place one has one token of color 7 and two of color 14; Place two one token of color 6 and one token of color 2. Since the arc expressions of the transition are satisfied, the transition is enabled. Fig. 2.7 shows the state of the CPN after the transition fired once. The tokens 7 and 6 are now at place three.

```
colset NUMBERS = int;
colset NUMBERSxNUMBERS = product NUMBERS * NUMBERS;
var number : NUMBERS;
var number2 : NUMBERS;
```

Listing 2.1: Definition of color sets

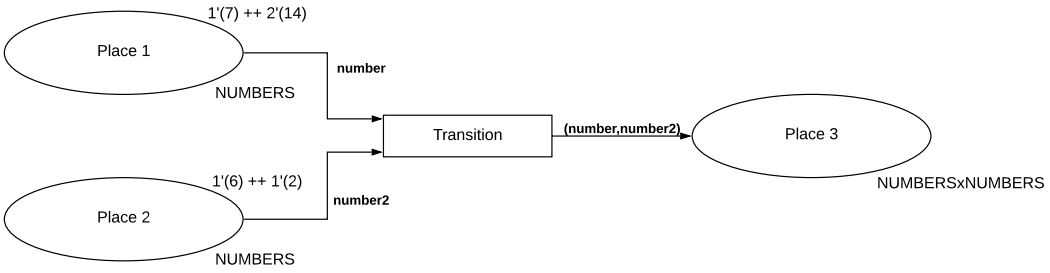

Figure 2.6.: Example of a CPN

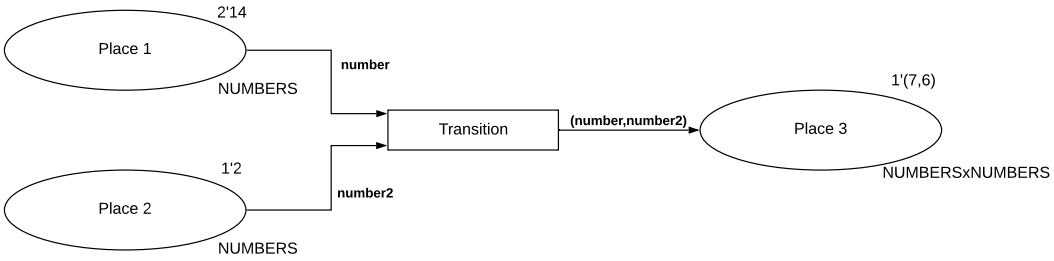

Figure 2.7.: State of the CPN after the transition fired once

2.6. Interoperability of Simulations

The Levels of Conceptual Interoperability Model (LCIM) [42] aims to provide a classification of systems regarding their level of interoperability. It defines five layers of interoperability from Level 0, where all data is only system-specific and there is no exchange of data at all, and Level 4, where the use and exchange of data are not only well-defined by

standard software engineering methods but there is also semantic consistency between the simulations.

Generally, the interoperability of simulations can be divided by three main concerns which are syntactical, semantic, and pragmatic interoperability [41]. While syntactic interoperability is about structuring the data in formats readable by other systems, semantic interoperability targets the meaning of data. Pragmatics is concerned with the purpose of exchanging the data. In order to provide meaningful interoperability, it is necessary to target it not only on the technical layer by enabling systems to simply exchange data via a common protocol but also on the semantic and pragmatic layer. This requires a conceptual alignment which describes *"the objectives, inputs, outputs, content, assumptions, and simplifications of the models [44]"*. For example, systems that interchange data about scheduling policies must agree about which policies are available and how they interpret them.

There are various approaches to generalized interoperability of simulations, e.g. the High-Level Architecture (HLA) which is also an IEEE standard [19]. However, they target interoperability at a very general level.

The simulation interface in this work will be designed specifically for the domains of the simulations to be coupled though it still aims to enable simple replacement of either one of the simulations.

3. Related Work

This chapter provides an overview of related work and distinguishes the approaches from the approach of this thesis. Using message brokers with message queues is a popular approach and various approaches to modeling them exist. Within ArchiMate [2], a modeling language for Enterprise Architectures, there are several options for modeling message queues. Clients can be connected to an application server by a so-called communication path that is annotated with *message queuing*, expressing that they communicate via messaging [8]. It is also possible to model messaging infrastructure in more detail by modeling queues and messages explicitly as first class entities [11]. However, ArchiMate does not support simulation or any other analysis of models.

Sachs et al. [35] use a methodology based on queuing Petri nets and provide a set of generic modeling patterns for modeling message-oriented systems. The approach and its predictions obtained by simulating the Petri net are validated with the SPECjms2007 benchmark. Self-adaptation is not considered.

Section 3.1 introduces already existing approaches to modeling message-driven systems with Palladio. Works on self-adaptation of systems using message queues are presented in Section 3.2.

3.1. Modeling Message-Driven System with Palladio

Model-driven performance prediction via parametric performance completions is proposed by Happe et al. [16, 17]. Performance completions aim to close the gap between a high level of abstraction in the model and the need for more detailed information in order to obtain accurate performance predictions by adding necessary refinement in automated model-to-model transformations. When designing a performance completion, possible factors affecting the performance are identified by performance analysts which then design experiments to evaluate their actual influence on the performance. To obtain quantitative information for a specific platform, the test-driver of the performance completion has to be executed. The obtained performance measurements can then be used to derive parametric resource demands for a platform-specific completion. The concept is illustrated by the example of modeling a message-oriented middleware with the Palladio Component Model where the resource demands are parameterized by e.g. the size of the messages and their arrival rate. This allows for considering the MOM as a black box and still obtain accurate predictions. Only the prediction of point-to-point communication is supported [17]. Parameter characterizations of a message can not be forwarded from the sender to the receiver [16].

Rathfelder et al. [31, 32] present an extension for modeling event-based interactions in Palladio. They model middleware as a reusable and parameterized component. Message

Property	Happe et al. [16, 17]	Rathfelder et al. [31, 32]	Czogalik [10]	This work
Calibration specific to	MOM, Hardware	Message types	MOM, Hardware	MOM, Hardware
Supported communication models	P2P	P2P, Pub/Sub	P2P, Pub/Sub	P2P, Pub/Sub
Components get notified upon arrival of messages	✓	✓	-	✓
Prediction of queue length	-	-	✓	✓
Increased latency for queued messages	-	-	-	✓
Flow control	-	-	-	✓

Table 3.1.: Comparison of the approaches for modeling message-driven systems using Palladio

queues are not explicitly modeled but abstracted [31, p. 108]. To simulate the model, they apply a transformation which converts the asynchronous events into synchronous calls and removes the event elements from the architecture. Resource demands are measured and modeled message-specific. They show the prediction accuracy for point-to-point and publish/subscribe communication in a case study.

Building on the work of Rathfelder, Czogalik [10] proposed modeling of a message-oriented middleware with consideration of queuing effects using the already existing elements in Palladio. Again, the model is reusable and parameterized. There exist different calibrations obtained from measuring different message brokers. In contrast to the approach of Rathfelder [31], resource demands are not modeled per message but derived by conducted measurements of a message broker dependent on the size of a message. In the simulation, a transformed model without the event elements is used. Receiving messages is modeled as a separate usage scenario. Even though the length of queues can be predicted, queuing effects do not increase the latency of messages. Queue lengths can not be limited and self-adaptations are not considered. The prediction proved to be accurate for point-to-point communication only.

Table 3.1 shows a comparison of the three presented approaches and the approach in this work regarding their properties. Though the approach of Czogalik [10] can predict the queue length, none of the presented approaches is capable of predicting an increased latency for queued messages or simulating flow control as opposed to the approach in this work.

Moreover, Klinaku et al. [24] demand for better modeling abstractions for message-driven systems to be offered in Palladio to obtain better predictions for elasticity, cost-efficiency, and scalability. In their case study, several scenarios that included e.g. message queues could only be modeled using workarounds. One was even impossible to model since it required self-adaptation rules.

Another extension of Palladio concentrating on events is the Indirections extension [20], though the focus there is on streams and aggregation of data. It extends the PCM with *DataSink* and *DataSource* roles that inherit from *ProvidedRole*, respectively *RequiredRole*. Several added actions enable the specification of new intracomponent behavior, such as, e.g., emitting and consuming of data. *DataChannels* transfer data between components. Additionally, they specify, among other things, how many elements are to be emitted via them, a scheduling policy, and a maximum length. Components can not only be notified when data is available but also poll for new data. The meta-model in this work is designed to allow for the integration of the Indirections extension.

3.2. Self-Adaptive Systems using Message Queues

Gotin et al. [13] conducted a case study on which performance metric should be used to auto-scale microservices that consume from message queues. Usage of queue-related metrics is compared to using CPU utilization as a metric. The scaling should prevent flooded or congested queues as well as overprovisioning of consuming microservices. A queue is considered as flooded when its length exceeds the maximum length a message broker can handle whereas congested means that the delivery times for messages are higher than desired. Consuming microservices are overprovisioned when they are allocated more resources than needed to process all incoming messages. Microservices are divided into two classes, compute- and I/O-intensive. It is shown that all metrics are suitable if the microservices have constant behavior. However, if there are changes in external services, queue metrics are better suited.

If it is not possible to provide more resources, overload protection by adapting the send rates is a possibility that Gotin et al. [14] presented. Send rates are adapted based on the current processing rate which is estimated based on queue metrics.

4. Modeling Message-Driven Systems

The main goal of this thesis is to support performance predictions for message-driven self-adaptive systems. For predicting the performance of a system at design-time, it needs to be modeled. To be able to model messaging middleware including queuing behavior on an architectural level, a meta-model for messaging is needed. A meta-model defines the structure of a DSML, as explained in Section 2.3. For capturing not only messaging aspects but also other properties of the system, integration into the PCM is useful. Therefore, some elements of the messaging meta-model inherit from or reference PCM elements. To extend PCM, the child extenders feature is used which allows for using the model elements of this work in Palladio models.

The modeled view types include the repository, SEFF, assembly, and allocation view type. The repository view type of this work enriches the PCM repository view type with the possibility to capture the ability of a component to send and receive messages and allows for the definition of routing keys. To be able to send messages, the extension to the SEFF view type defines a new action to send messages with a probabilistic routing key. Within the assembly view type, we offer new model elements to connect components sending and receiving messages via messaging infrastructure such as message channels, routers, and brokers. New messaging allocation contexts then allow for the allocation of these new elements in the allocation view type.

Routing specifics are included in the assembly view type. We also considered to model the routing in a specific message routing view type and connect the communicating components directly in the assembly view type. However, this was discarded since it would have increased the modeling effort: One of the original goals of using messaging is to decrease the coupling between components, i.e. a message-sending component does not need to know about the eventual destination of the message. Connecting components directly in the assembly view type would break that. It would also be complex to ensure consistency between both view types since the message routing view type must connect the components over possibly several channels and routers as specified in the assembly view type. Additionally, endpoints are channel-specific [18, p. 96] and can, therefore, be connected to one channel only. If several components are connected with another, one needs to think about how communication should happen in terms of channels anyways to figure out how many endpoints are needed.

In the following, Section 4.1 describes the elements of the meta-model. Possible reconfigurations of a system and their implications on the model are considered in Section 4.2. An approach to formalization of the semantics of the meta-model using Colored Petri Nets is given in Section 4.3.

4.1. Meta-Model elements

The meta-model is organized in the four view types already mentioned above: repository, assembly, allocation, and SEFF. Whenever possible, we used inheritance from elements of the PCM as an extension mechanism to avoid duplicate modeling and provide easy integration. The meta-model in this work is about messaging specifics only whereas the other aspects of a system can be modeled using the PCM elements. The general structure of the meta-model is based on the patterns of Hohpe and Woolf [18] that Section 2.1 presented.

To illustrate the usage of the model elements, a simple alarm notification system will be modeled throughout the section. In this system, one component sends alarm notifications from two different countries, Germany and France. Another component receives alarm messages and processes them. This component will be instantiated two times: One instance is responsible for processing alarm notifications from Germany and the other one for the notifications from France.

4.1.1. Repository

To model repositories, a *MessagingRepository* is needed as a root element since routing keys are introduced as new first-class modeling elements directly contained in the messaging repositor. This means that routing keys exist independently from potential use by other elements. However, the *MessagingRepository* inherits from the Palladio repository and therefore offers all Palladio repository elements for modeling. In the long term, not both are needed and the additional features of the messaging repository could be merged into the Palladio repository. Since messaging is a form of event-based communication, *SendEndpoints* and *ReceiveEndpoints* inherit from *SourceRole*, respectively *SinkRole*. The *EventGroup* of a *MessagingEndpoint* defines the supported *MessageTypes* of it. For containment reasons, we introduced a *MessagingComponent*, which inherits from *BasicComponent*, to allow for the creation of messaging endpoints. *MessageTypes* inherit from *EventType* and additionally allow for the definition of a root routing key. *RoutingKeys* are *Entities* and can be hierarchic. The term root routing key means that a message type could have either that routing key or a routing key which has the specified root routing key as a parent over possibly several levels. An example of a hierarchy of routing keys is the routing key *AlarmFR* whose parent routing key is *Alarm*. The full routing key can be displayed as *Alarm.AlarmFR*.

An already existing alternative to that is using variable characterizations and then branch depending on their value. However, characterizations only allow for string values. Having a model element for routing keys makes modeling more type-safe and also has a clear semantics. Fig. 4.1 shows a class diagram of the added elements. Inherited properties are greyed out.

Fig. 4.2 depicts the repository of the alarm notification system as an example. All elements are contained in a messaging repository unless depicted otherwise. It contains the two messaging components *Transformator* and *Extractor*. The *Transformator* provides the operation interface *ISendAlarms* and can also send messages of the *Alarms* event group via its send endpoint. These messages can be received by the *Extractor* component via its

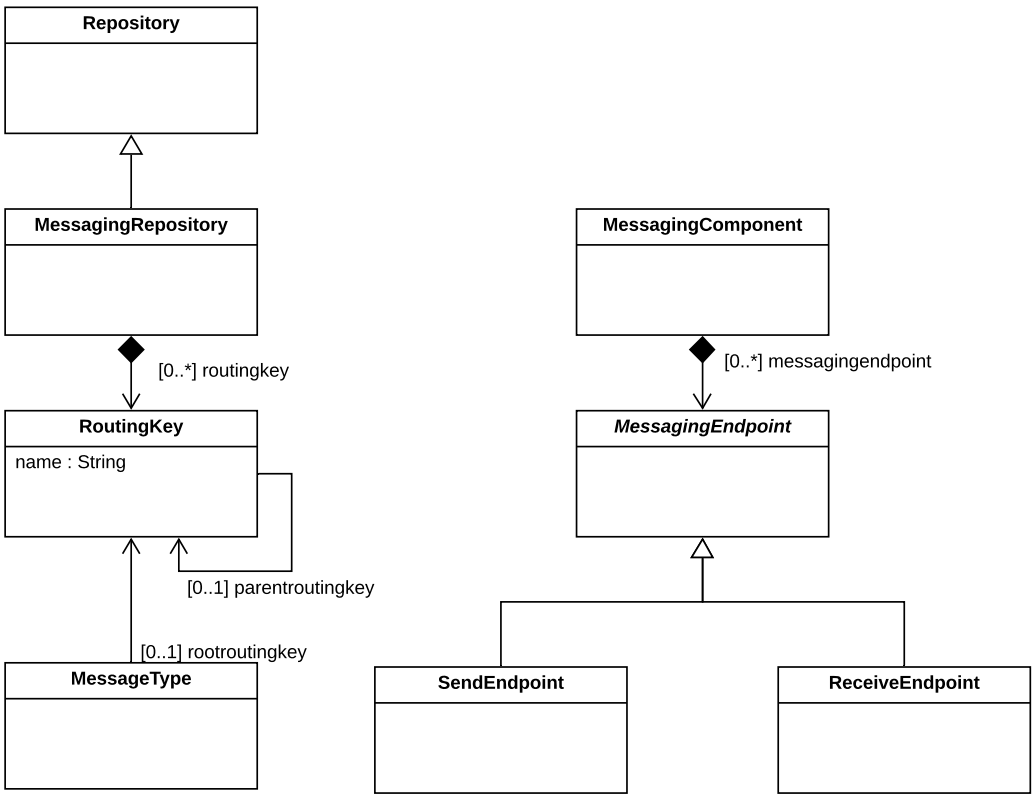

Figure 4.1.: Elements of the meta-model of the repository view type

receive endpoint. The message type *Alarm* is contained by the *Alarms* event group and has the root routing key *AlarmRK*. The routing keys *AlarmDE* and *AlarmFR* specify the *AlarmRK* as their parent routing key.

4.1.2. Service Effect Specification

In the SEFF view type, we added a new action called *SendMessageAsync* for sending messages. It inherits from *AbstractAction* and *CallAction*. Analogously to the *EmitEvent* action it has a reference to the send endpoint used and the message type sent. The routing keys of the messages can be specified by adding *ProbabilisticRoutingKeys* which each reference a routing key and specify its frequency as double. To determine the routing key of a message in the simulation, an EnumPMF is built out of all probabilistic routing keys specified as it will be explained in more detail in Section 5.5. Messages having one routing key only can be modeled by setting that routing key as the only probabilistic routing key of the action with a frequency of 1. Fig. 4.3 depicts the added action and its properties while Fig. 4.4 shows a SEFF model for the send endpoint of the *Transformator* component presented in the previous section using the *SendMessageAsync* action. When the `sendAlarm(String msg)` method provided is called, the SEFF starts and executes the *SendMessageAsync* action. The action sends an alarm message with the probabilistic

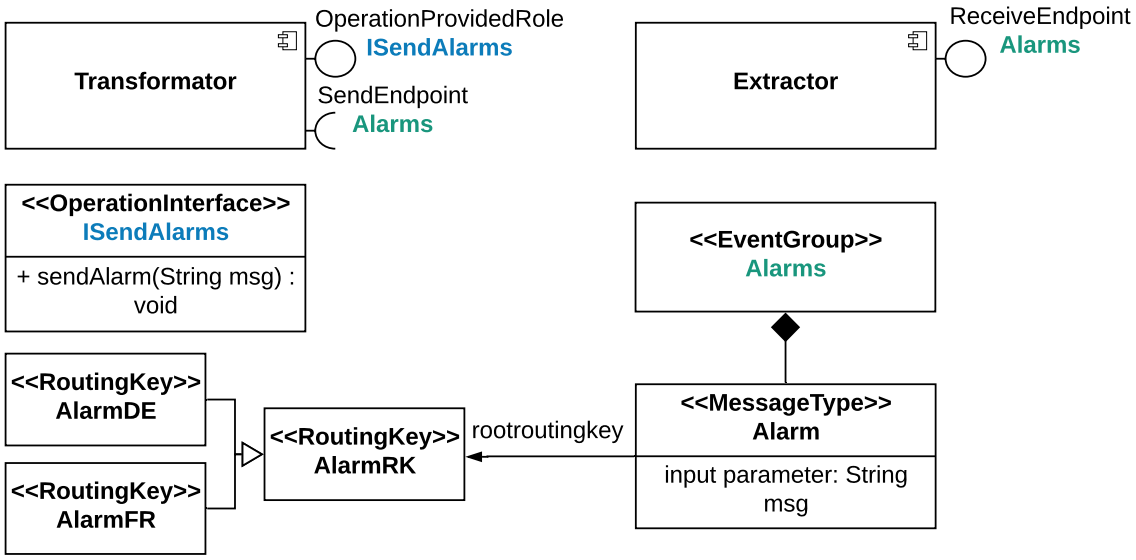

Figure 4.2.: Repository of the alarm notification system

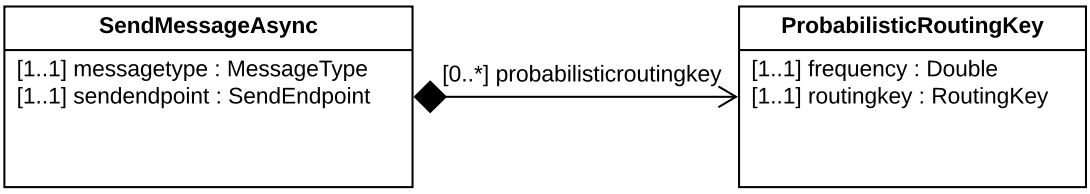

Figure 4.3.: The *SendMessageAsync* action

routing keys *AlarmDE* and *AlarmFR*. The size of the message sent is characterized by a *BYTESIZE* characterization and is the same as the *BYTESIZE* of the input parameter.

The modeling alternative here was to use variable usages and directly specify an EnumPMF as variable characterization. However, EnumPMFs support only strings as literals whereas with the *ProbabilisticRoutingKey* feature it is possible to refer to routing keys already present in the model. This eases modeling for the user. We also considered adapting the syntax of stochastic expressions. This was disregarded because of the potentially high effort needed and the limited benefit for the contributions of this work.

Receiving a message is specified by simply selecting the corresponding connected receive endpoint in a RDSEFF.

4.1.3. Assembly

To model assemblies that use messaging, we introduce a new root element called *MessagingAssembly*. A *MessagingAssembly* directly contains *MessageChannels*, *RoutingEntities*, and *MessagingDelegationConnectors* and inherits from the PCM *System*. We decided to

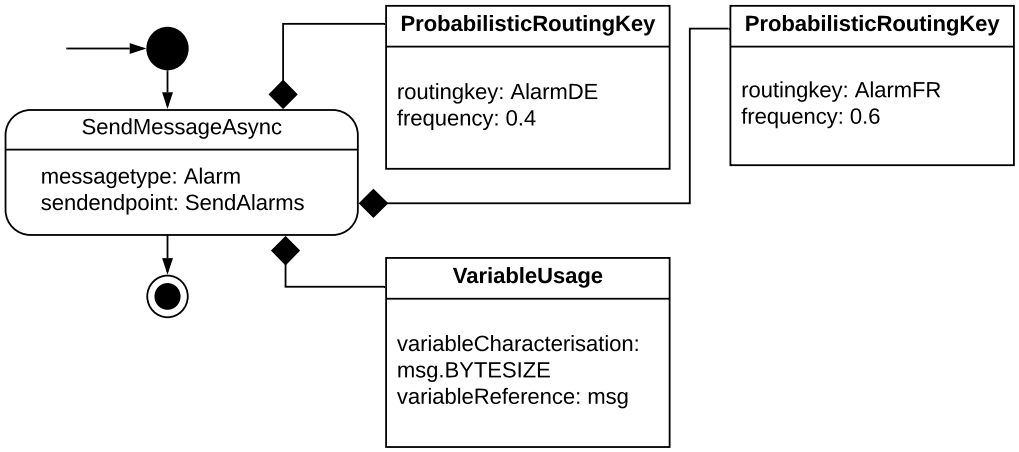

Figure 4.4.: SEFF model using the new *SendMessageAsync* action

include routing entities in the assembly view type since we assumed that they are not very complex and unlikely to be reused. *RoutingEntities* can be either a *MessageRouter*, *MessageAggregator*, or a *MessageBroker* which contains further routing entities. This enables the modeling of message broker hierarchies. Routing entities connect to *DatatypeChannels* via *ConnectedSendEndpoints* and *ConnectedReceiveEndpoints* which are channel-specific and have an event group. They are required to have at least one sending and one receiving endpoint since otherwise, they can not actually route something. Message routers and aggregators also have an enum *RoutingStrategy* which at the moment supports the literal *contentbased* only. This realization makes use of the strategy pattern which suggests to not implement an algorithm directly into a class but to store a reference to an object realizing it. This way, different variants of the algorithm can be used easily. In this context, this refers to the message routers and aggregators not directly containing a specific strategy of routing but only storing a reference to the *RoutingStrategy* enum. This enables changing the routing strategy without the need to select a different type of router or aggregator.

Enums do have the disadvantage of difficult extensibility, however, child extenders are allowed to extend the *RoutingStrategy* enum. Fig. 4.5 depicts the different routing entities and their relations. The endpoints of a message broker can be connected to the endpoints of another routing entity with *MessagingDelegationConnectors*. They have the same semantics as Palladio delegation connectors which can not be used because they connect operation provided roles and need a reference to an assembly context.

MessageChannels can be *Point-to-Point*, *Publish/Subscribe* or *DeadLetter* and are unidirectional. Every channel has a reference to *QueuingProperties* which specify the properties of the resulting queue in the simulation which are maximum length, if the queue should be durable, and a scheduling policy which determines in which order messages are taken out of the queue and forwarded to consumers. At the moment, Round-Robin and First-Come-First-Serve can be chosen as a scheduling policy. The scheduling policy is modeled as an enum and can be extended by child extenders. A *DeadLetterChannel* is a special case of a message channel and must be contained by a routing entity. If an incoming message can

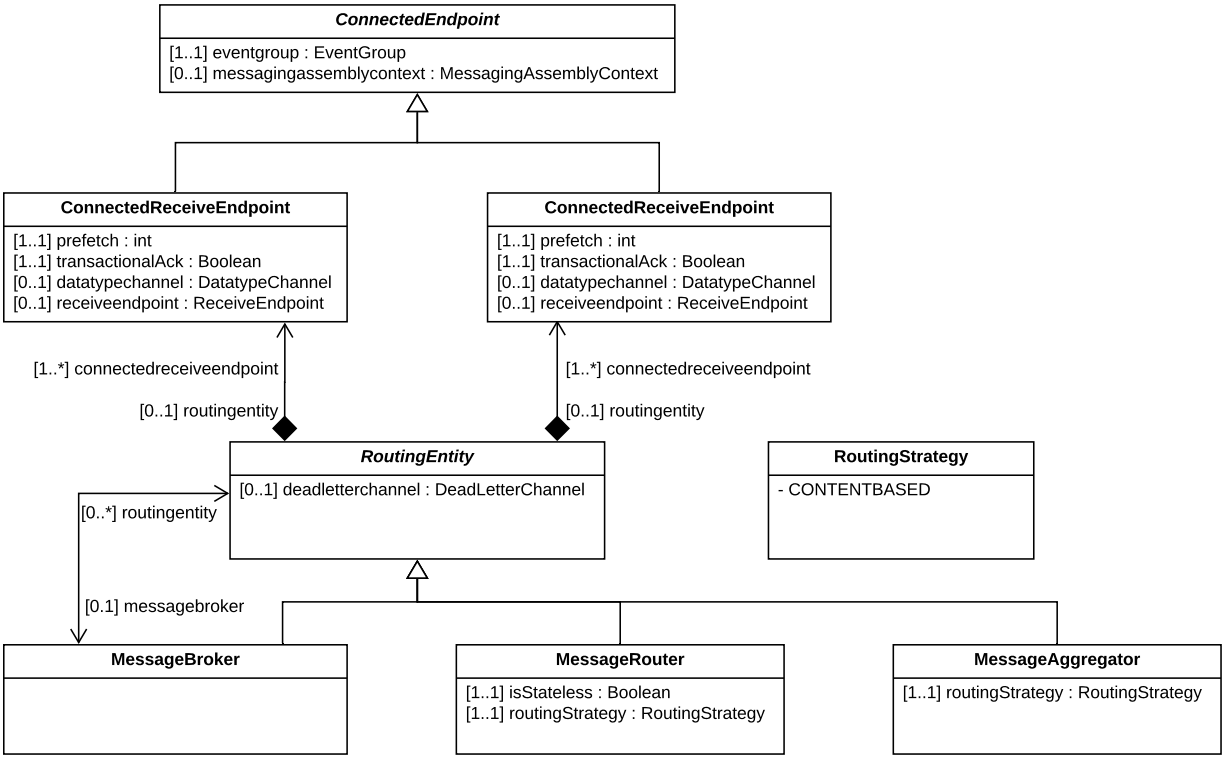

Figure 4.5.: The available routing entities and their relations

not be routed to any outgoing message channel, the message is forwarded to the dead letter channel. Point-to-point and publish/subscribe channels are *DatatypeChannels* which have an event group and can have multiple connected send and receive endpoints. However, they are required to have at least one sending and one receiving endpoint connected. The event groups of the connected endpoints must be the same as the channel event group. To enable routing based on the routing key, information on which routing key is accepted needed to be integrated into the assembly view type. Therefore, datatype channels can optionally specify an accepted root routing key. Messages with routing keys differing from the accepted root routing key and not having it as parent routing key will be discarded or routed to a dead letter channel, if present. The hierarchy of message channels and their properties can be seen in Fig. 4.6. The routing key could have also been integrated to connected receive or connected send endpoints. However, that would arise the question on how to deal with different connected endpoints specifying different routing keys connecting to the same channel. In the case of connected receive endpoints, an accepted routing key would even seem like a message filter which it should not be. Associating this information to datatype channels makes clear which behavior can be expected from it when connecting to it and eases modeling. This equals a contractual specification: If a connected endpoint of a compatible event group to the datatype channel connects to it, the endpoint can send, respectively receive, messages to or from it. Compatible here

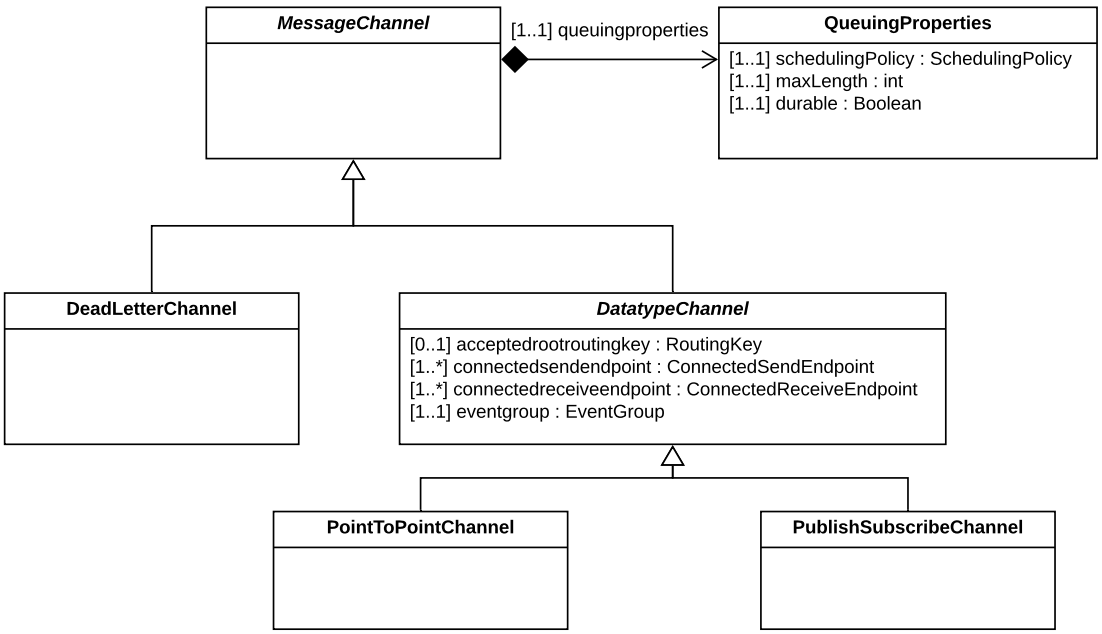

Figure 4.6.: Hierarchy of message channels

means that the event group of the endpoint should be the same as the event group of the channel, or inherit from the channel event group.

MessagingEndpoints defined in the repository are instantiated with *ConnectedEndpoints* contained by *MessagingAssemblyContexts* in order to connect to message channels. *ConnectedReceiveEndpoints* additionally specify a prefetch which indicates how many messages can be sent to it without having acknowledgments for them, and if the acknowledgment sent should be transactional, i.e., is sent before or after processing the message at the receiving side. This differs from the usual Palladio modeling where roles do not need to be instantiated. However, source and sink roles, from which the messaging endpoints are inheriting, must refer to an *InterfaceRequiringEntity* or *InterfaceProvidingEntity*, respectively. Routing entities can not inherit from that since it would allow for the modeling of non-messaging related concepts, e.g., operation interfaces and signatures. Therefore, a separate concept for the modeling of endpoints of routing entities is needed. To unify the modeling in the assembly view type, we decided to use *ConnectedEndpoints* for routing entities as well as for messaging assembly contexts.

An alternative approach supporting the usual Palladio modeling would have been to introduce connected endpoints for routing entities only, and use the messaging endpoints for connecting to channels from messaging assembly contexts. However, this has the disadvantage of having two different model elements representing the same concept which could be confusing for users. Additionally, specifying the prefetch and acknowledgment behavior of a consumer in the assembly view type would only be possible by defining component parameters in the repository view type and setting them in the assembly view type. However, component parameters do not have a clear semantics using them this way.

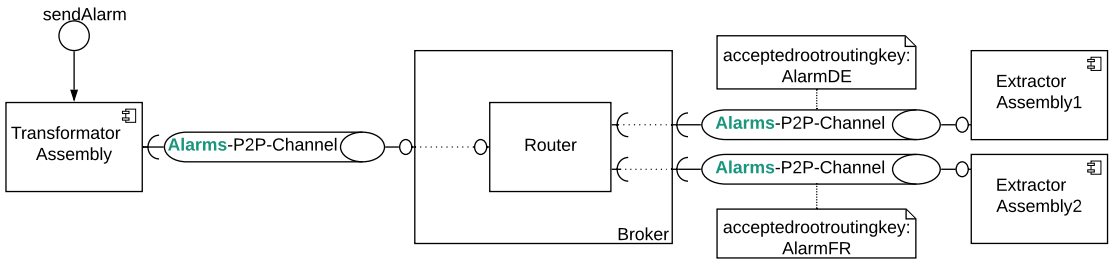

Figure 4.7.: Assembly model of the example alarm notification system

Moreover, they would overwrite the values of variables inside the respective component with the value set in the assembly view type when, e.g., a variable in the SEFF of the component has the same name as the component parameter.

Fig. 4.7 shows an assembly for the example alarm notification system, however, some details as, e.g., the queuing properties of the channels and prefetch of connected receive endpoints are omitted. The *TransformatorAssembly* has its connected send endpoint connected to a Point-To-Point channel with the event group *Alarms*. The *Router* receives messages from that channel; its connected receive endpoint is connected to the channel via a receive messaging delegation connector. The connected send endpoints of the router are connected to two Point-To-Point channels via send messaging delegation connectors. Both channels have the *Alarms* event group, however, they define different root routing keys: While *ExtractorAssembly1* will only receive messages with an *AlarmDE* routing key, *ExtractorAssembly2* gets all messages with *AlarmFR* as routing key.

4.1.4. Allocation

Since the Palladio allocation context has a constraint that either an assembly context or an event channel must be allocated, a *MessagingAllocationContext* is needed for allocating message channels and routing entities which can not inherit from the Palladio allocation context because of that constraint. A new constraint ensures that a message channel or a routing entity is referenced. Therefore, a new *MessagingAllocation* must contain the new messaging allocation contexts and can inherit from the Palladio allocation.

4.2. Replications of Elements due to Reconfigurations

The presented meta-model should be appropriate for self-adaptive systems and support reconfigurations. Hence, we discuss possible replications of elements due to self-adaptations of the system or reconfigurations made from the outside, and their mapping to the model in the following. There are two main reasons for replicating elements which are load-balancing and failover. The modeling presented in this work is supposed to cover load-balancing scenarios. Failover configurations are not explicitly considered because they are not part of the architectural model. However, recovery from faults can be modeled using Palladio recovery blocks.

Considered scenarios for load-balancing are the replication of producers, consumers, and routers. We do not consider the replication of single channels because it is not a common case and very technical. The scenarios are illustrated starting from the base scenario in Fig. 4.8. When a producing assembly context P is replicated, the replicated assembly context P' can simply connect its connected send endpoints to the same channels as P is connected to. Fig. 4.9 covers that replication scenario.

Nearly the same applies when a consuming assembly context C is replicated: If the assembly context is connected to a point-to-point channel, the replicated assembly context C' can simply connect its connected receive endpoint to the same channel C is connected to. The point-to-point property then ensures that the message is delivered exactly once. Fig. 4.11 depicts this. In the case of a publish/subscribe channel, replicating a consumer for load-balancing is not a very likely case. Fig. 4.12 shows the base scenario for that. However, it can be mapped to the model by adding a new point-to-point channel with the same properties as the publish/subscribe channel and connecting the connected endpoints of C and C' to it, as shown in Fig. 4.13. The same holds in the case of router replication: If the connected receive endpoints of a router are only connected to point-to-point channels, the replicated router R' can connect its connected endpoints to them and the load will be balanced between both routers. Fig. 4.10 illustrates that. In the case of publish/subscribe channels, it can be decided whether only the connected receive endpoints connected to point-to-point channels are replicated or if new point-to-point channels are established as for the consumer replication in case of publish/subscribe channels. In any case, the connected send endpoints of R' can connect to the channels R is sending to.

Figure 4.8.: Base scenario for replications

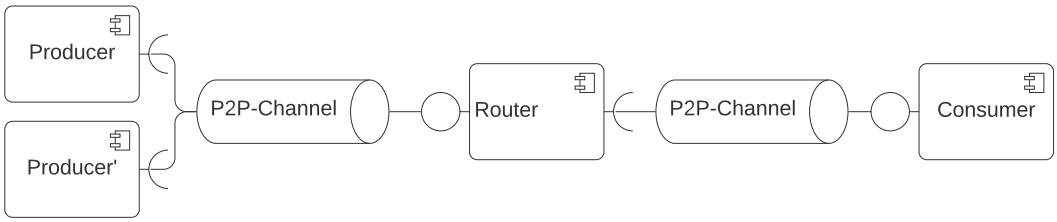

Figure 4.9.: Replication of producer

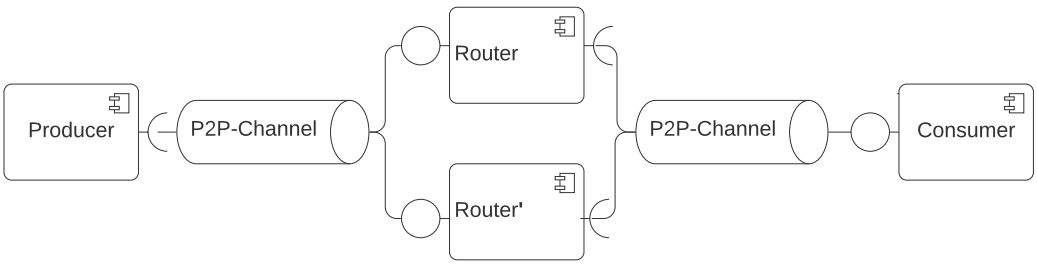

Figure 4.10.: Replication of router

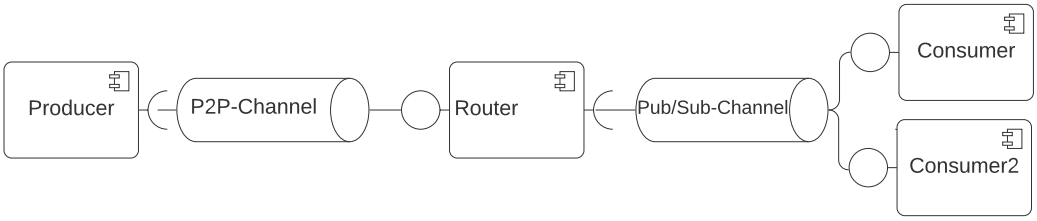

Figure 4.11.: Replication of consumer

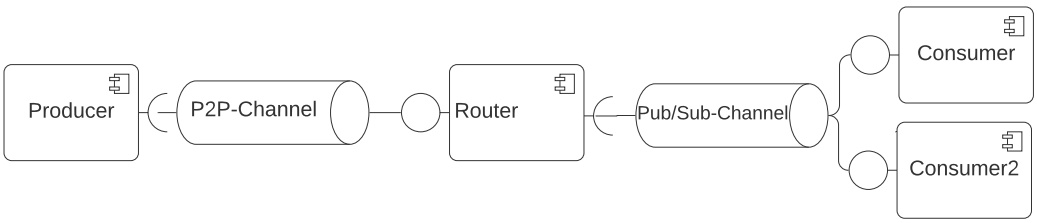

Figure 4.12.: Base scenario with a publish/subscribe channel

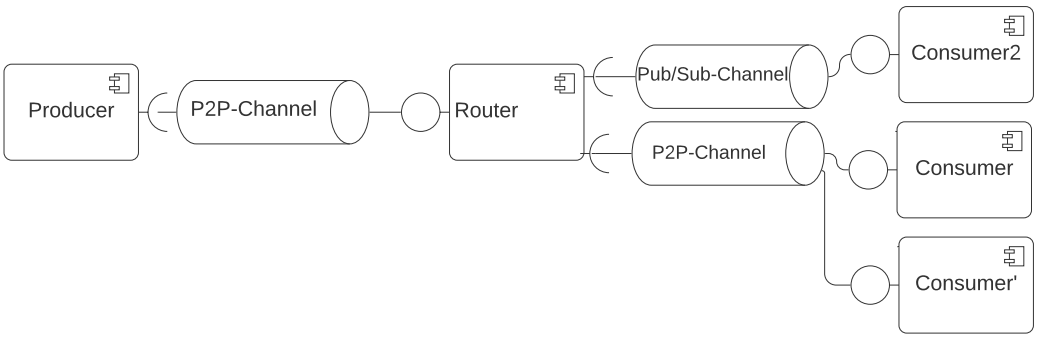

Figure 4.13.: Replication of consumer in case of a publish/subscribe channel

4.3. Formalization using CPNs

A formal representation of the semantics of the metamodel can be given using Colored Petri Nets (CPNs) as explained in Section 2.5. Formalizing semantics has the advantage of capturing it unambiguously. Therefore, models have a more precise meaning. Furthermore, the properties of a system can be proved. While a mapping for the main concepts will be given, modeling aspects such as reliability or timing are not in the scope of this work.

We model messages as tokens of a color set that is a product of the two color sets *group* and *payload*. Group represents the event group as in the meta-model, whereas payload is just an int value specifying the size of the message. The exact data contained in the message is not relevant for the functionality of the system and is therefore abstracted, however, the size of a message does influence, e.g., its latency. Different event groups can be represented using different colors: Every event group is represented by a unique color. The definition of these colors in CPN ML, as introduced by Jensen and Kristensen [23], is presented in listing 4.1. An enum list defines the available event groups. An example token of color *GROUPxPAYLOAD* is, e.g., (*ALARMDE*, 7) where *ALARMDE* is the color for the event group "alarms from Germany" and 7 the size of the message.

```
colset GROUP = with ALARMFR | ALARMDE;
colset PAYLOAD = INT;
colset GROUPxPAYLOAD = product GROUP * PAYLOAD;
```

Listing 4.1: Definition of color sets

Places depict applications or routing entities. Applications may have initial tokens if they send messages, routers wait for incoming tokens and forward them. Aggregators only forward messages if they have received a sufficient number of messages. If an application or a routing entity sends messages of various event groups via different send endpoints, the transition has a guard accepting only tokens of the right color, respectively event group. Message types and routing keys are not considered here but could be integrated by defining additional color sets for each of them and including them in the arc expressions of transitions. This causes the transition to be enabled only when the group and the routing key, respectively message type, are conforming to the arc expression.

Transitions and their in- and outgoing arcs represent message channels. A point-to-point-Channel includes several transitions from one place, all accepting the same event group. There is only one outgoing arc from each transition to a different consumer. This ensures the property that every message is consumed by exactly one consumer since tokens can not replicate in a place: Even though all transitions are enabled, only one of them can fire and forward the token to a consumer. Fig. 4.14 shows a sketch of a CP-Net realizing a point-to-point channel: When a token, that binds to the type of the variable message, arrives at Component 1, all transitions are enabled but only one of them will fire. A publish/subscribe channel includes only one transition with several outgoing arcs, one to every consumer subscribed to the channel. Since outgoing arcs create tokens, every consumer receives the message when the transition fires. Fig. 4.15 illustrates the concept. Incoming messages are queued at routing entities by using lists as the accepted datatype

of routing entities. Ingoing transitions append to the list whereas outgoing transitions remove the first element.

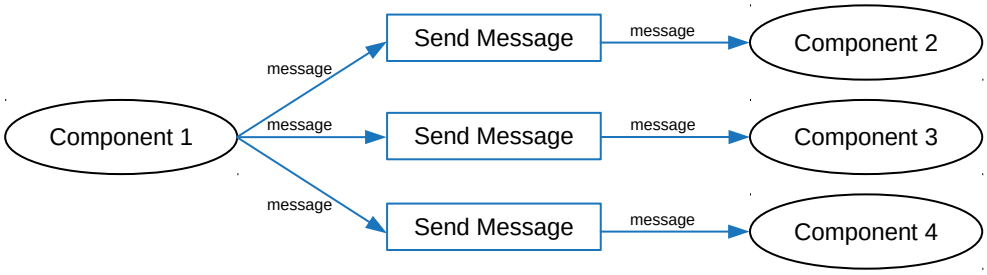

Figure 4.14.: Point-to-point-Channel

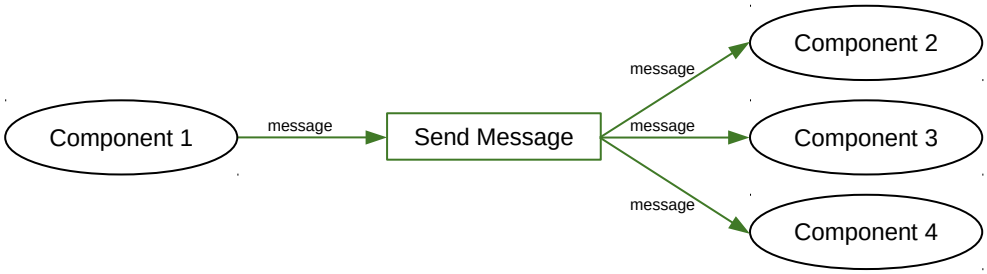

Figure 4.15.: Publish/subscribe Channel

Fig. 4.16 sketches Round-robin scheduling for a point-to-point channel. To model scheduling, new *scheduling* places are introduced. The *scheduling* place that should enable the first transition gets one *RR* token in the beginning. Since now all transitions of the channel require a *RR* token to be enabled, only that can fire. When the transition fires, the *RR* token is passed to the next *scheduling* place which will forward the token to the next transition until the token arrives at its first place again, and the whole process begins all over. Fig. 4.17 shows a point-to-point channel that uses Round-robin scheduling as CPN with initial tokens at Component 1 and the upper scheduling place. The color sets from Listing 4.1 are used, as well as the color sets and variables shown in Listing 4.2. The enabled transition is marked green. Because the *SendMessage* transitions need a token of type $GROUPxPAYLOAD$, as well as an *ACK* token, only the upper transition is enabled. When the transition fires, one of the $(AlarmFR, 7)$ tokens is sent to Component 2 and the transition below gets activated as the *ACK* token is transferred to the scheduling place below. Fig. 4.18 shows the CPN after the first transition has fired. When now the lower *SendMessage* fires, the *ACK* token is transferred to the upper scheduling place and thus enabling the upper transition again. Fig. 4.19 depicts the state of the CPN after the lower transition fired.

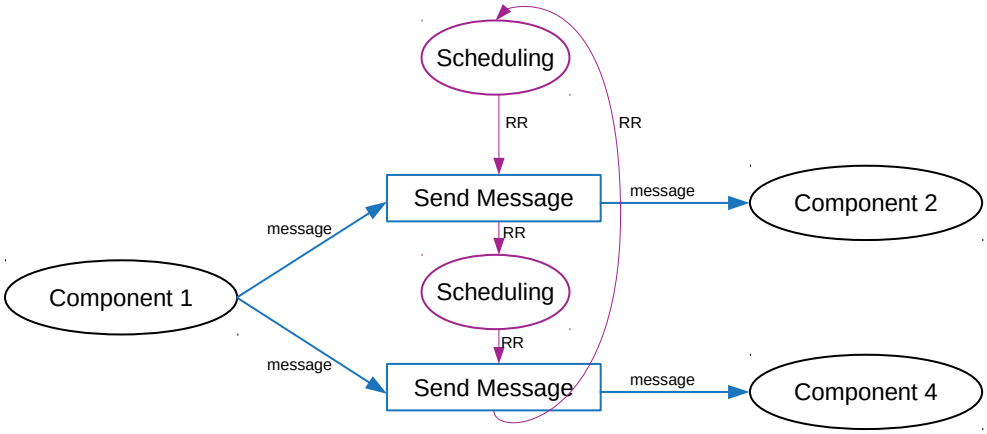

Figure 4.16.: Round-Robin scheduling (violet) combined with a Point-To-Point channel (blue)

```
colset ACK = with acktoken;
var RR : ACK;
var message : GROUPxPAYLOAD;
```

Listing 4.2: Definition of color sets for Round-Robin scheduling

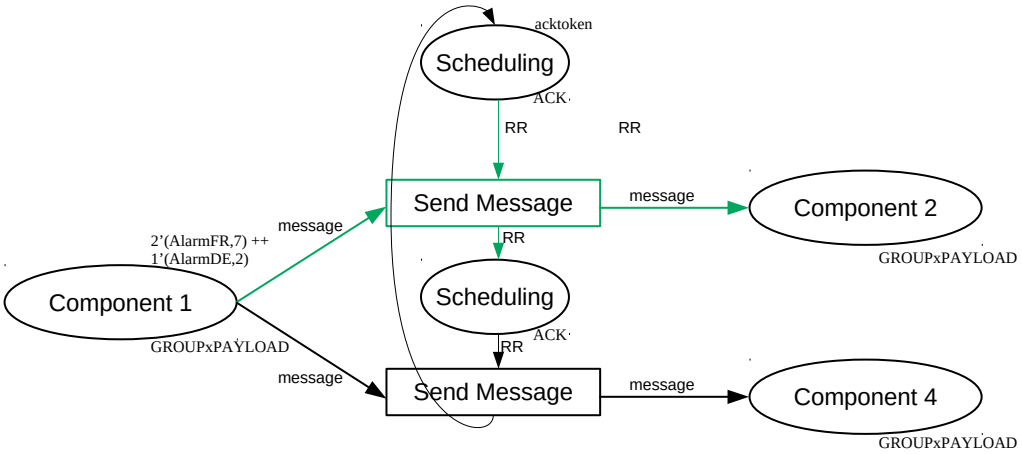

Figure 4.17.: Execution of a CPN using Round-Robin for scheduling (initial marking)

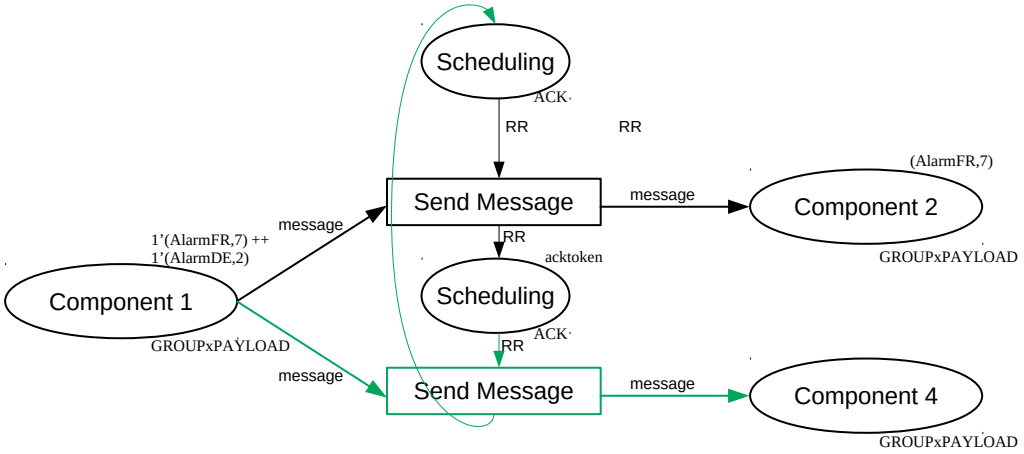

Figure 4.18.: Execution of a CPN using Round-Robin for scheduling (Step 1)

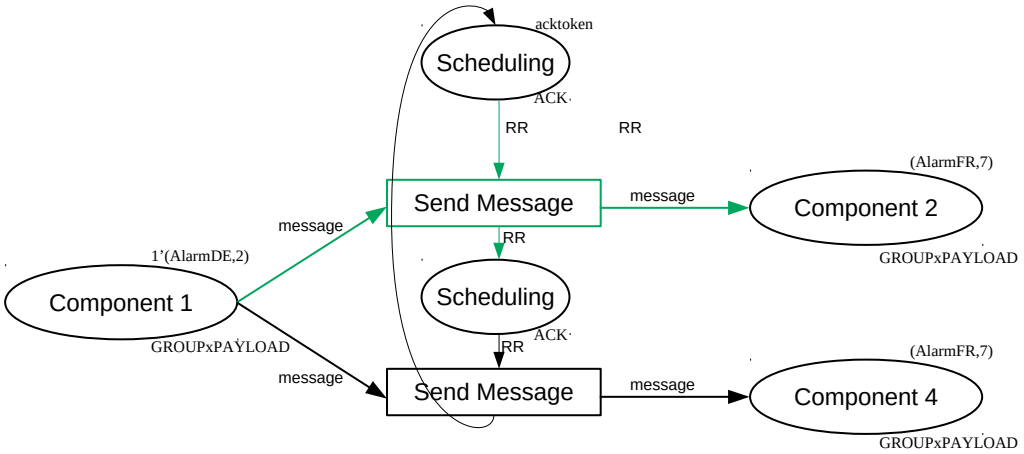

Figure 4.19.: Execution of a CPN using Round-Robin for scheduling (Step 2)

Dead letter queues can simply be modeled by adding a transition with the inverse of the union of all other transition guards from outgoing transitions of that place as transition guard. This transition then leads to a place simply collecting unroutable messages. For example, when the color sets from Listing 4.3 are used and a router only routes messages of the group *ALARMFR*, the dead letter queue transition would define a guard that the group of the message must be *ALARMDE*, because that is the inverse of the union of all transition guards outgoing from that router.

Prefetch of consumers and acknowledgments can be mapped by *ack* tokens of an acknowledgment color set. Fig. 4.20 depicts their integration with point-to-point channels and Fig. 4.21 with publish/subscribe channels. The acknowledgment part is colored orange. In both cases, there are as many *AckPlaces* of the sending component as there are receiving components. They have a certain amount of initial tokens which determine the prefetch of the consumers. In the case of point-to-point channels, every consumer can specify a different prefetch. For publish/subscribe channels, the *AckPlace* with the least initial tokens will determine the prefetch of all consumers. The *SendMessage* transition can only fire if there is an *ack* token in the respective *AckPlace* needed. For point-to-point channels, this is the case per transition to one consumer; however, for publish/subscribe channels, it is required that all *AckPlaces* have at least one *ack* token. The first *SendMessage* transition makes a tuple out of the *ack* and the *message* token. They are now in a new *In-Delivery* state that accepts the product of the color sets of *message* and *ack*. When now the next *SendMessage* transition fires and the message is delivered to the consuming component, the acknowledgment is delivered back to its *AckPlace*, thus giving the sender new credit.

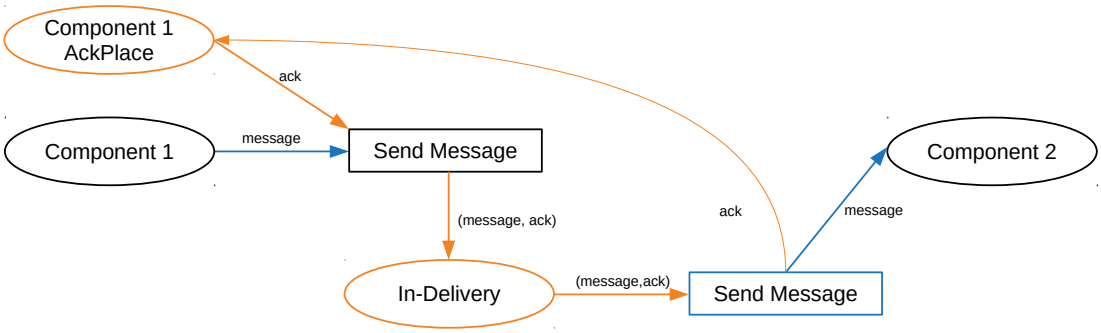

Figure 4.20.: Acknowledgments (orange) combined with a Point-To-Point channel (blue)

Fig. 4.23 shows an example system modeled using the model elements from the presented meta-motel. Modeling details that are not mapped to the CPN model are omitted, e.g. queuing properties. *AssemblyContext1* sends messages to the *Router* via a point-to-point channel their connected send and receive endpoints are connected to. The router routes the messages to *AssemblyContext2* and *AssemblyContext3* via two different channels depending on their event group: Messages of event group A are routed via the point-to-point channel A to *AssemblyContext2*, messages of event group B via the point-to-point channel B to *AssemblyContext3*.

Fig. 4.22 depicts the same system modeled as CPN. The colors used are consistent to the colors of Fig. 4.23. Listing 4.3 shows the color sets and variables used in this example.

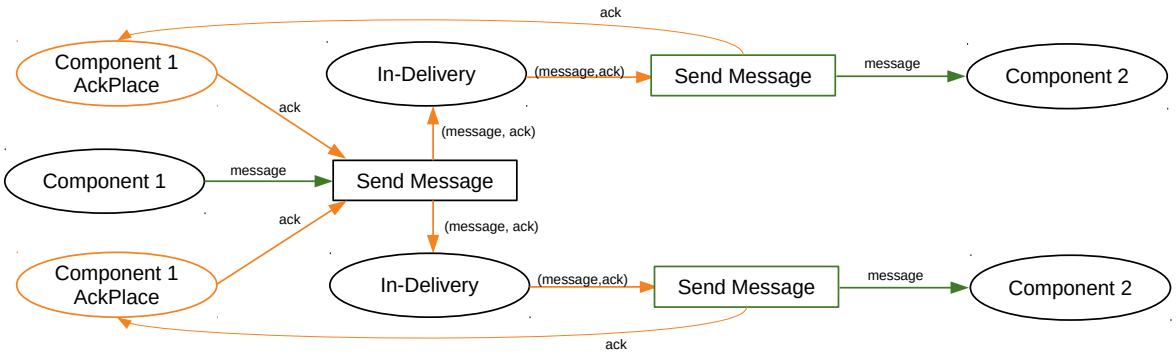

Figure 4.21.: Acknowledgments (orange) combined with a Publish/Subscribe channel (green)

```
colset GROUP = with A | B;
colset PAYLOAD = INT;
colset GROUPxPAYLOAD = product GROUP * PAYLOAD;
colset MESSAGEQUEUE = list GROUPxPAYLOAD;
var messagequeue : MESSAGEQUEUE;
var group : GROUP;
var payload : PAYLOAD;
```

Listing 4.3: Color sets of the example system

Component 1 sends messages via a point-to-point-Channel to the *Router*, where incoming messages are appended to the message queue. They are routed to different channels depending on their group: If the group of a message is A, the *Receive Messages A* transition fires and delivers the message to *Component 2*. If the group of the message is B, the *Receive Messages B* transition fires and delivers the message to *Component 3*.

Fig. 4.24 depicts another, simplified version of the system that uses an aggregator as a routing entity. The additional variables *group*2 and *payload*2 are defined similar to *group* and *payload*. The red part of the arc expression makes up the aggregator functionality: While the *msgqueue* variable can be an empty list, the *Receive Messages* transition needs at least two messages present at the aggregator to fire.

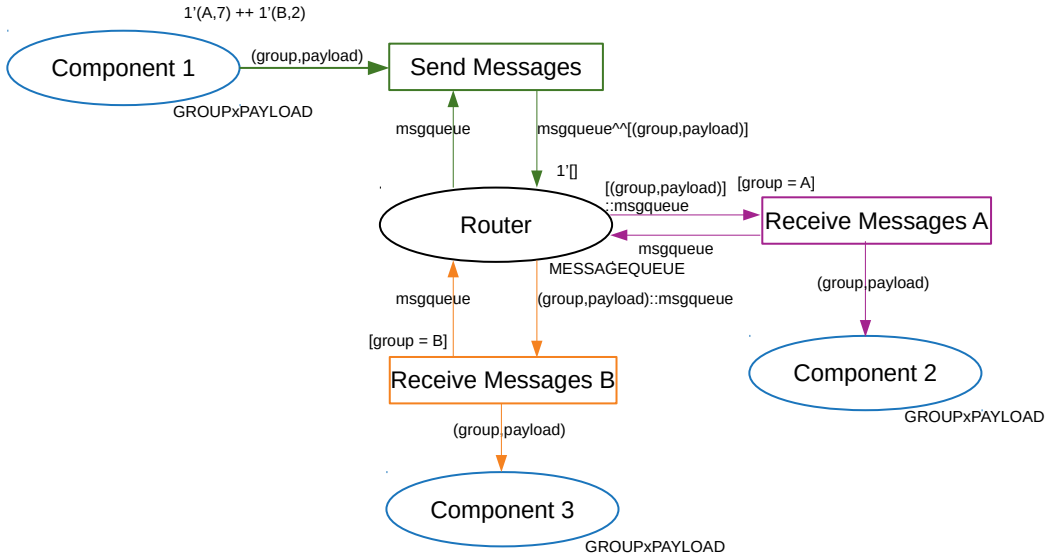

Figure 4.22.: An example system modeled as CPN

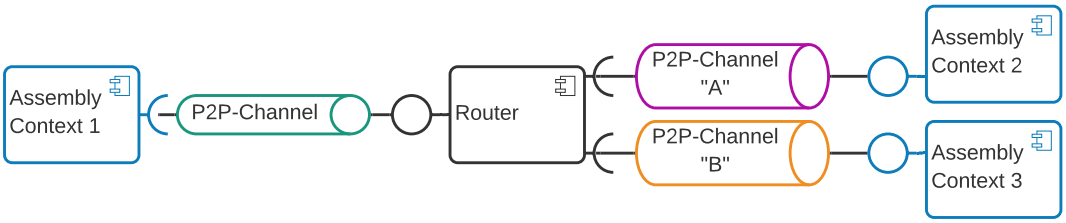

Figure 4.23.: An example system modeled using the elements from the presented meta-model

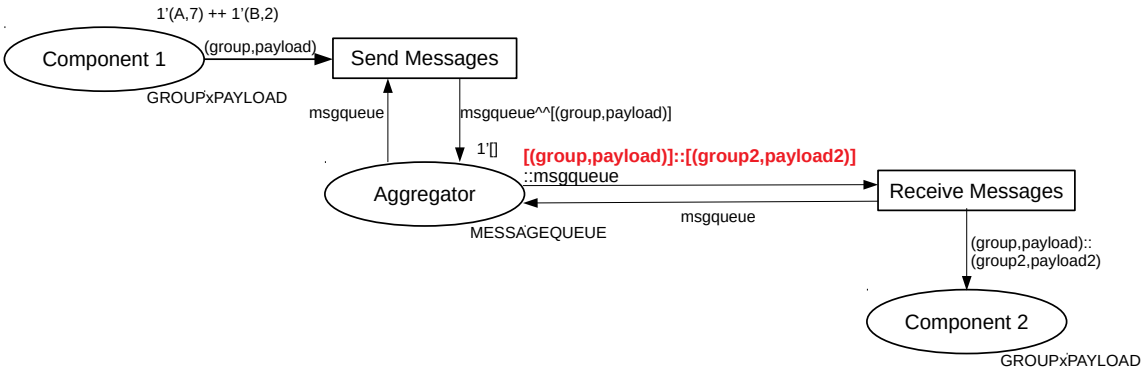

Figure 4.24.: An example system as CPN using an aggregator

5. Simulating Messaging

To simulate component-based systems using messaging as a communication paradigm, the existing simulation SimuLizar will be combined with the messaging simulation introduced in Section 2.1.4 through a simulation interface. We chose SimuLizar because message queues are a common method to achieve decoupling in dynamic systems that are reconfiguring themselves.

Though we already gave a formalization for the semantics of the developed meta-model and highlighted advantages of formal models, we will not perform formal verification of system models to derive quality predictions but simulate them.

Theorem proving and state-space methods are the main approaches for verifying a formal model and its properties [43]. While system properties can be proven by theorem proving, this method is not suitable for answering questions about how a system behaves if it is not clear what that behavior exactly is. It can only be proven that a system has a certain, formulated behavior. Moreover, it does not give any hints about why a certain proof does not succeed [43]. This makes it not a good choice for getting information about systems and their behavior at design time. The state-space method has the advantage of providing counterexamples if a property does not hold [22, p.7]. However, the state explosion problem [43] makes the analysis of even relatively small systems infeasible: The state-space method is based on exploring all possible states of a system that may be very high or even infinite [22, p.7] very fast. Furthermore, formal analyses are static program analyses that do not execute programs. This does not allow for changes in the system during an analysis which is necessary for predicting the quality of self-adaptive systems.

First, the requirements for a general simulation interface between a simulation of a component-based architecture description language (ADL) and a messaging simulation are introduced in Section 5.1. Based on these requirements, we developed a concept that is presented in Section 5.2. The implementation for SimuLizar and the messaging simulation as introduced in Section 2.1.4 will be described in Section 5.3. Section 5.4 covers the integration of measurements. Transformation of the Palladio modeling elements to the elements of the broker simulation is explained in Section 5.5. How self-adaptations can be specified based on measurements taken during a simulation run and how they affect the entities in the messaging simulation is explained in Section 5.6.

5.1. Requirements for a Simulation Interface

A simulation interface between a simulation of a component-based ADL and a messaging simulation must ensure technical as well as semantical interoperability. On the technical side, both simulations should have the same timeline to ensure consistency and synchronization between them. Semantically, concepts and requests from one simulation must be

mapped to the entities of the other simulation. While enabling that, re-use and replacing of one of the simulations should still be as easy as possible. This should also ease the separate testing of both simulations.

5.2. Concepting a Simulation Interface

As already remarked in Section 2.6, it is necessary to target standardization at the modeling level to ensure meaningful interoperability as some issues regarding, e.g., semantics and different structures for the same concept can not be dealt with on the technical level [42]. Therefore, we developed a generalized concept of a simulation interface between a component-based ADL and a messaging simulation and provide a mapping of the respective concepts. Followed by that, we give an overview of the realization in this work.

As stated in Section 5.1, the component simulation and the messaging simulation should be as independent of each other as possible. Therefore, the concept of the simulation interface needs the component simulation to know about where a message is addressed to only and requires no further knowledge of the messaging infrastructure. On the other side, the messaging simulation only gets the size and the routing key of a message and is not interested in the content of the message sent. When a message is sent, the simulation interface is invoked and stores the content and context information of a message by a unique message ID assigned to it. It only forwards the size and the routing key, as well as the message ID, to the producer in the messaging simulation corresponding to the component sending the message in the component simulation. The behavior of the sending component can then be executed further while the message is in delivery in the messaging simulation. When a message arrives at a consumer, the simulation interface gets notified with information about the consumer and retrieves the stored information about the message by its ID. The information about the consumer identifies the receiving component whose behavior when receiving this type of message is then triggered with the actual content of the message. Acknowledgments of consumers are triggered by the component simulation because the messaging simulation should not be aware whether acknowledgments are transactional or not. When a message should be acknowledged, the component simulation must give information about the receiving component and the ID of the message to acknowledge. The simulation interface then sends this request to the messaging simulation with the ID of the consumer corresponding to the component information. On the semantic side, the simulation interface must, therefore, provide a mapping between components and producers, respectively consumers to realize its functionality as explained above. An overview of the mapping of entities and requests between the simulations is provided in Tab. 5.1.

Technical interoperability should be provided by simulating both models in one simulation. This gives them both the same timeline with the same future event list.

This concept comes with the advantage of asynchronous sending behavior: Senders consider the sending process finished after passing a message to the simulation interface and return immediately in terms of simulation time. Also, no usage scenarios need to be specified for consuming messages: Components receiving messages are notified upon arrival and the execution of their respective behavior is then triggered. Moreover, both

Component simulation	Messaging simulation
Component sending messages	Producer
Component receiving messages	Consumer
Component sends a message	Corresponding producer sends message
Corresponding component receives a message	Consumer receives message
Component sends acknowledgment for a message	Consumer sends acknowledgment

Table 5.1.: Overview over the mapping of entities and requests between a component and a messaging simulation

simulations are transparent to each other. It is therefore easy to replace either one of them; messaging simulations only need to provide a method for sending messages and acknowledgments via a specified producer, respectively consumer, and a callback providing information about the receiving consumer.

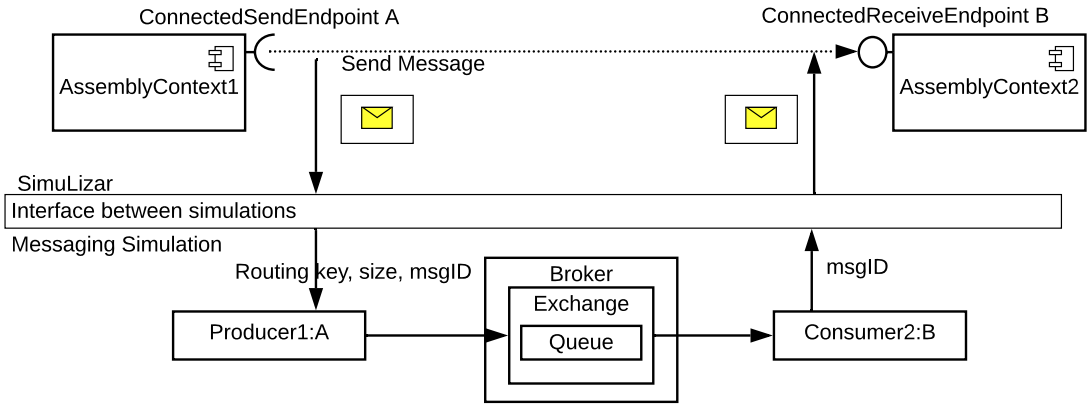

Figure 5.1.: Overview of the simulation interface concept

An overview of the realization of the concept for SimuLizar and the AMQP messaging simulation as described in Section 2.1.4 is given in Fig. 5.1. The order of events is from the top left via the bottom left and bottom right to the top right. When an assembly context wants to send a message via a connected send endpoint, the routing key of the message is determined via the parameters of the *SendMessageAsync* action, as it was introduced in Section 4.1.2. After the message and its routing key are passed to the simulation interface, SimuLizar considers the action done and moves on to the interpretation of the next action. The simulation interface is responsible for storing the Palladio-relevant message information and handing the size, routing key, and ID of the message as well as information about the producer, which should send the message, to the messaging simulation. The corresponding producer is determined by the assembly context sending the message and the send endpoint used. The producer in the messaging simulation then

sends the message. When a message arrives at a consumer, the simulation interface is notified with information about the consumer and the ID of the received message. The information about the consumer identifies the corresponding assembly context and its receive endpoint. The RDSEFF of it can then be triggered with the stored palladio-relevant information of the message.

5.3. Implementation

We decided to extend SimuLizar via the *modelobserver* extension point because classes contributing to that extension point are initialized before the simulation starts and observe the Palladio model, which is both needed for initializing the messaging model and adapting it to possible changes. It would have also been possible to directly extend the *SimuLizarRuntimeState* class, which provides access to all simulation and SimuLizar related objects and keeps the current simulation state. This would have meant to provide a dedicated "messaging analysis" or to directly modify the SimuLizar code to launch the extended *SimuLizarRuntimeState*. However, semantically, simulating messaging is not a new analysis but only an extension to the existing analysis. Therefore, we decided to implement the extension using the extension point. Fig. 5.2 gives a general overview of the structure.

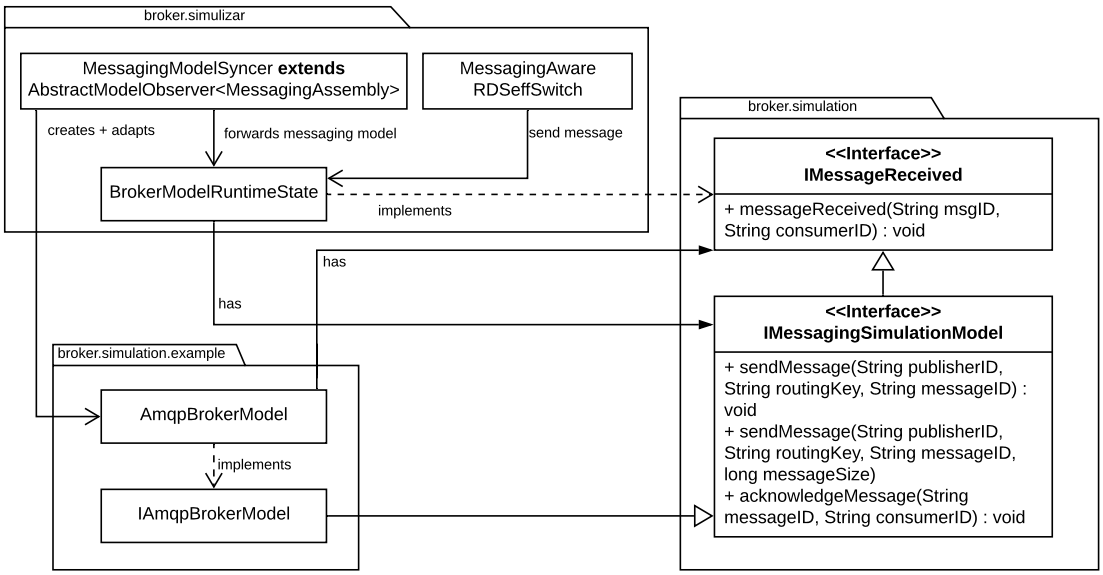

Figure 5.2.: Overview of the implementation structure

When executing the *MessagingModelSyncer*, the messaging model is created as an *IAmqpBrokerModel* based on the Palladio model. The details of the transformation will be explained in Section 5.5. However, to enable callbacks from the entities held in the *AmqpBrokerModel* to it, it implements the *IMessageReceived* interface so entities do not need to know about the type of simulation model they belong to.

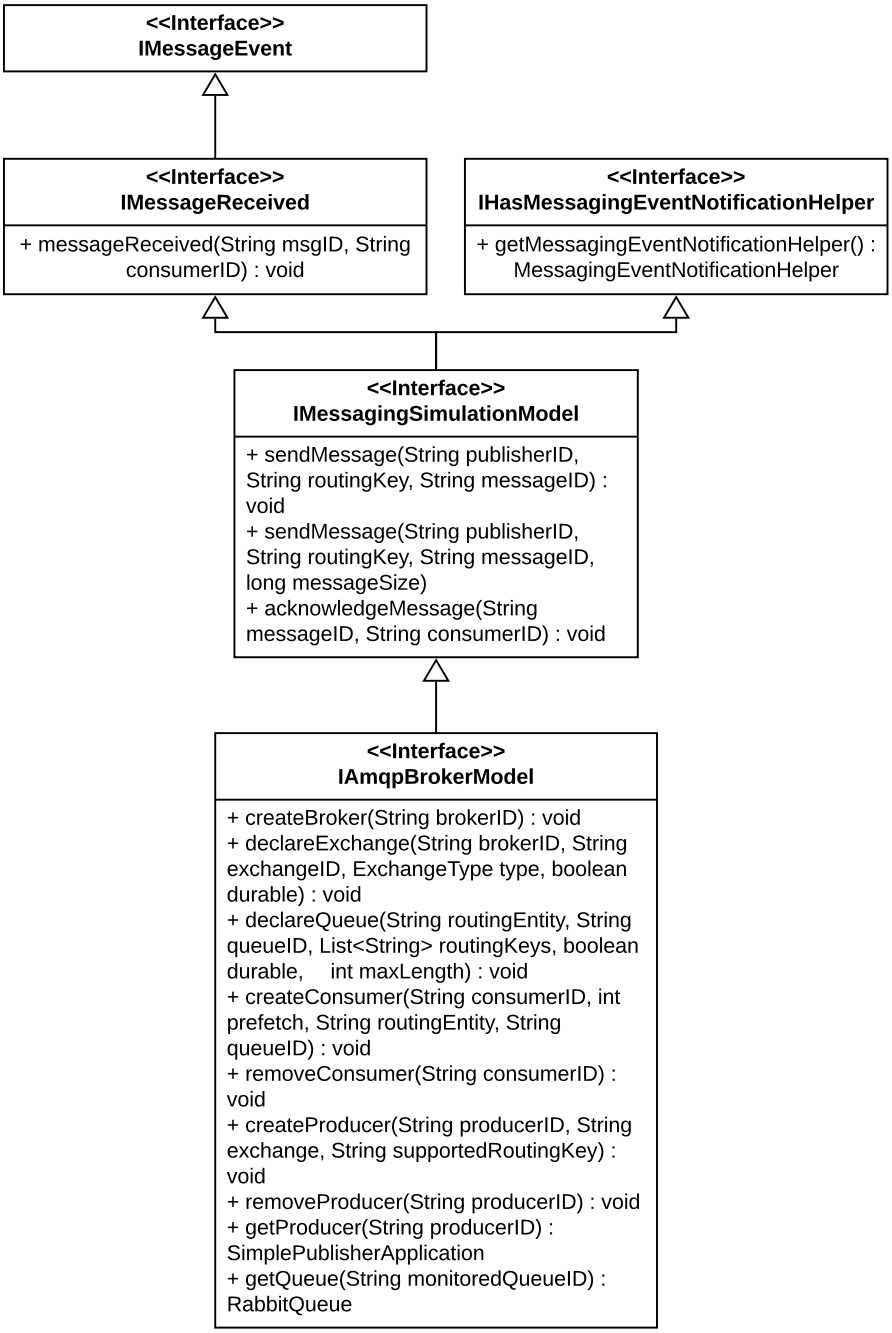

Figure 5.3.: Overview of the interface structure

To realize the requirement of the same timespan for both simulations, the simulation control and simulation engine factory of the SimuCom model are also set for the messaging model. The SimuCom model is the simulation model used by SimuLizar for simulating users, system behavior, and resources. This way, they both have the same simulation timespan and events of both simulations are scheduled in the same list of future events.

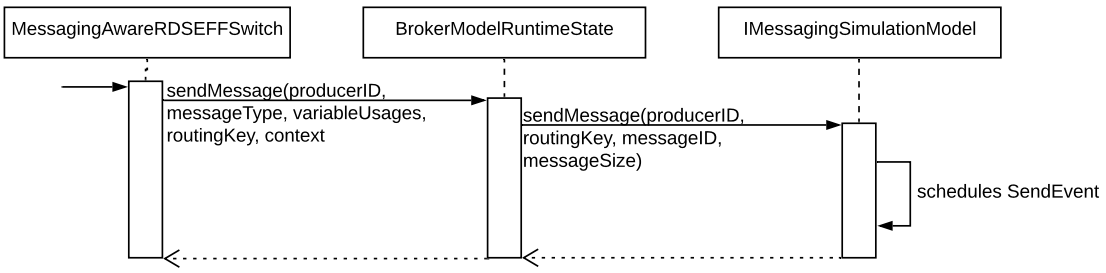

Figure 5.4.: Process of sending a message

Desmo-J, which is one of the simulation engines used for executing the simulation, also allows for setting submodels of a model. However, other simulation engines do not support this which led to the decision of using the same simulation control for both models.

When the *AmqpBrokerModel* is created, the *BrokerModelRuntimeState* is set as its *IMessageReceived* attribute and it is forwarded to the *BrokerModelRuntimeState* that stores it as *IMessagingSimulationModel*. The hierarchy of interfaces can be seen in Fig. 5.3. Whenever a *SendMessageAsync* action is about to be executed, the *MessagingAwareRDSeffSwitch* is called via the *rdseffswitch* extension point from SimuLizar. The *RDSeffSwitch* of SimuLizar interprets resource demanding SEFFs. The *MessagingAwareRDSeffSwitch* then determines the routing key of the message by evaluating the *ProbabilisticRoutingKey* property of the *SendMessageAsync* action. The property is evaluated by building a stochastic expression: The individual *ProbabilisticRoutingKeys* constitute an EnumPMF with their name as an enum and their probability as the probability of the respective enum.

This expression is then evaluated and the full routing key is determined by concatenating the probabilistic routing key with its potential parent routing keys and the event group of the message. Assembly context and send endpoint ID are used for constructing the name of the respective producer in the messaging simulation. Routing key and producer id are then passed to the *BrokerModelRuntimeState* together with other relevant information associated with the message, which are message type, input variables, and the execution context.

The *BrokerModelRuntimeState* determines the size of the message to send via a *BYTESIZE* variable characterization if present. Otherwise, a standard size will be assigned by the messaging model. The other information is stored until the arrival of the message. The message is then sent via the send message method of the messaging model. Fig. 5.4 shows this process.

When a message arrives at a consumer, the *AmqpBrokerModel* is notified and itself notifies its *IMessageReceived* reference, which is the *BrokerModelRuntimeState*. Based on the ID of the consumer, the receiving Palladio assembly context and the receive endpoint is determined. The other message information is retrieved by the ID of the message. Execution of the respective RDSEFF in Palladio is triggered by creating and scheduling a *ConsumerProcess*, inheriting from *SimuComSimProcess*, which uses the same request context as for sending the message but a copy of the stored context for execution. It also gets the assembly context, receive endpoint, and message type. With this information, a

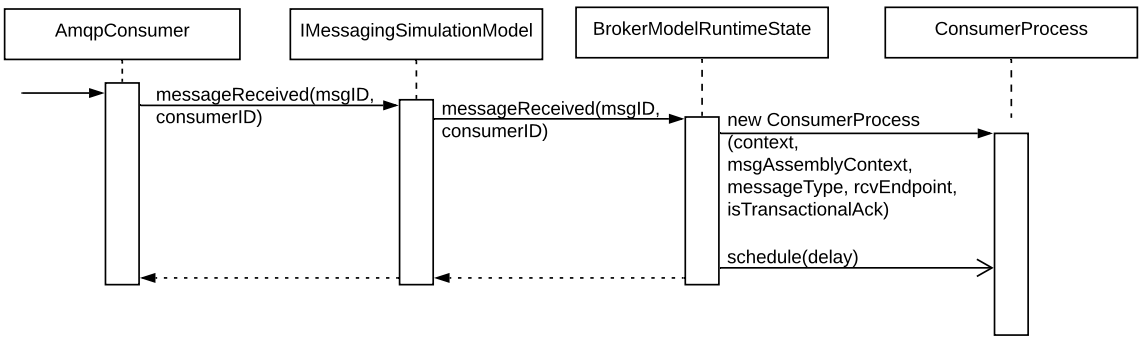

Figure 5.5.: Process of receiving a message

RepositoryComponentSwitch is created and the `caseProvidedRole` method called with the receive endpoint as parameter. Depending on whether the acknowledgments should be transactional, they are sent before or after that. Fig. 5.5 depicts the process of receiving a message until the creation of the consumer process.

5.4. Integrating Measurements into the Messaging Simulation

To measure metrics that are specific to messaging, they must be captured by the simulation. Therefore, we integrated and extended the Quality Analysis Lab (QuAL) [25]. Considered metrics in this work are latency of messages, queue length, queue input rates, queue growth, and memory consumption of queues. However, the implemented integration can easily be extended to support further metrics.

We integrated the metrics by creating a new library of metrics descriptions. The measurement of the new model elements is enabled by newly created measuring points: *ReceiveMessageMeasuringPoints* are attached to a messaging endpoint and an assembly context. *MessageLatencyMeasuringPoints* reference a message type of which the latency should be measured. Message channels and their resulting queues can be measured by a *QueueMeasuringPoint*. In the case of measuring a Publish/Subscribe-Channel, a connected receive endpoint must be given to determine the correct queue to be measured.

Fig. 5.6 gives an overview of the structure of the implementation. The *BrokerProbeFrameworkListener* creates probes and calculators according to the monitors defined in the monitor repository model. For the building of calculators, the *IGenericCalculatorFactory* of the probe framework context of the SimuCom model is used for ensuring that the calculated values are persisted by the Experiment Data Persistency & Presentation (EDP2) framework. The *BrokerProbeFrameworkListener* is also responsible for taking measurements whenever an event is fired by the observed event notification helpers. Therefore, it implements the two interfaces *IBrokerInterpreterListener* and *IMessagingInterpreterListener* where the former observes a *MessagingEventNotificationHelper*, responsible for events in the SimuLizar environment such as response times of receive endpoints and latency of

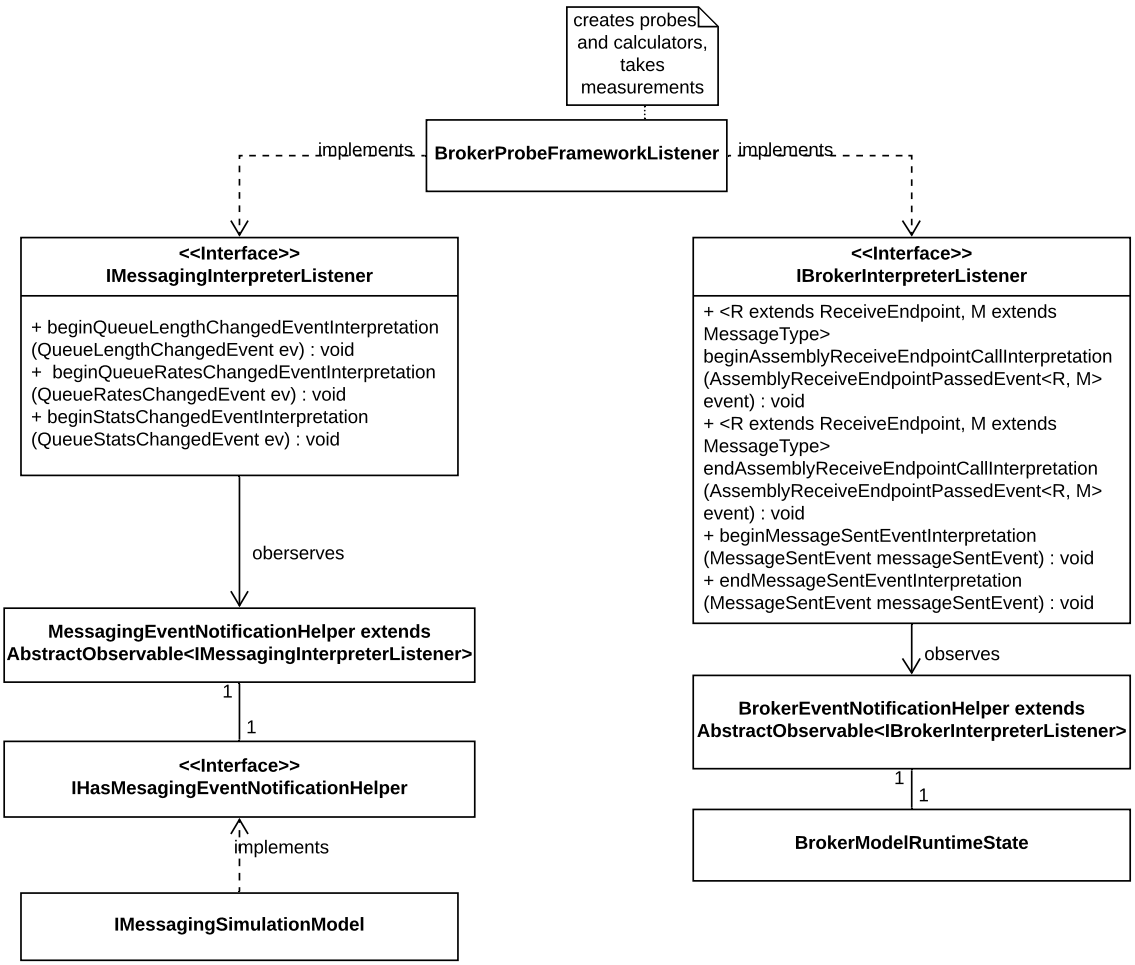

Figure 5.6.: Structure of measurements integration

messages of one message type, and the latter a *BrokerEventNotificationHelper*, responsible for events in the messaging simulation.

The elements in the messaging simulation are aware of the existence of the messaging event notification helper only and create and fire the respective events via their reference to the *IMessagingSimulationModel* that every element of the messaging simulation has. Supported events are the *QueueLengthChangedEvent*, *QueueRatesChangedEvent* and *QueueStatsChangedEvent*.

Events concerning the latency of messages and the response time of receive endpoints are fired by the *BrokerModelRuntimeState*. When the latency of a message should be measured and it is delivered to several consuming assembly contexts, e.g. via a Publish/Subscribe-Channel, only the first time of delivery is taken into account for calculating its latency.

Meta-model element	AMQP entity
Connected Send Endpoint	Producer
Connected Receive Endpoint	Consumer
Point-to-point channel	One queue
Publish/subscribe channel	One queue per Connected Receive Endpoint connected to the channel
Dead Letter channel of a router	Queue without consumers
Broker	Broker
Router	Topic exchange

Table 5.2.: Mapping of meta-model elements to AMQP entities

5.5. Transformation

For the simulation of the newly introduced model elements, they need to be transformed into entities of the messaging simulation. As introduced in Section 2.1.4, a RabbitMQ inspired messaging system that adheres to the AMQP 0.9.1 protocol is simulated here. To enable an easy mapping from the model elements to the created AMQP entities and vice versa, the id of the respective model name is set as the id of the AMQP entity it is transformed to. Tab. 5.2 gives an overview of the mapping of meta-model elements to AMQP entities.

First, broker nodes are created as specified in the model. Every router is translated to a topic exchange, located on its given broker or a default broker otherwise. Dead letter channels are created as dead letter queues; dead letter channels located on brokers are not considered because messages in RabbitMQ are always addressed to an exchange and can therefore not be identified as undeliverable at brokers. Dead letter queues are ordinary queues but do not have regular consumers consuming from it.

For every connected receive endpoint, one consumer is created having the specified prefetch and consuming from the queue which was created based on the datatype channel the endpoint is connected to. For publish/subscribe channels, one queue for every consumer is created. point-to-point channels result in only one queue. The queue is bound to the exchange with the accepted root routing key of its corresponding channel and has the queuing properties as specified in the model. A message is routed to a queue when its routing key starts with or is equal to the routing key of the queue. For example, when a queue is bound with the routing key *Alarms*, messages with the *Alarms* or *Alarms.AlarmsFR* routing key will be routed to it.

A producer is created for every connected send endpoint of an assembly context. It publishes to the exchange or exchanges that has or have connected receive endpoints connected to the datatype channel of the connected send endpoint the producer is corresponding to. If the connected receive endpoint belongs to a broker, the messaging delegation connectors are evaluated. To account for the accepted root routing key of the channel, producers only send a message if it starts with or is equal to that. Queuing properties of channels from producers to brokers or exchanges are not transformed.

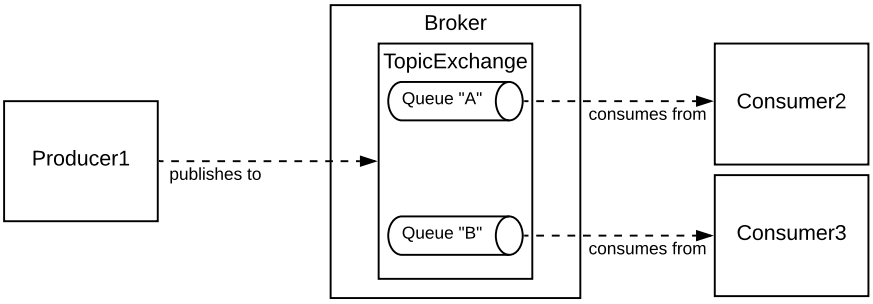

Figure 5.7.: Example of a transformed model with a point-to-point channel

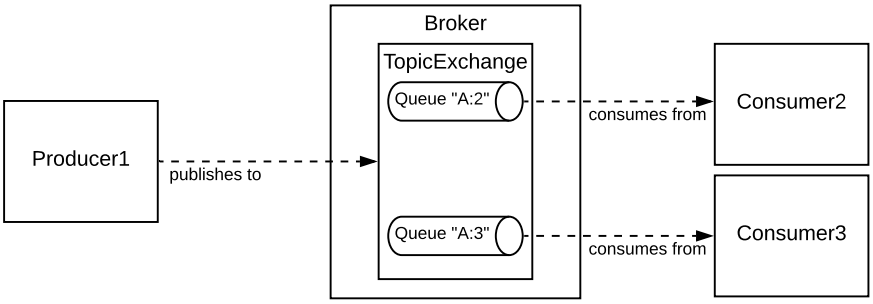

Figure 5.8.: Example of a transformed model with a publish/subscribe channel

Fig. 5.7 gives an example of the system in Fig. 4.23 after the presented transformation to AMQP entities was applied. The sending and receiving assembly contexts result in producers and consumers, in- and outgoing point-to-point channels with the same event group result in one queue located on a topic exchange, which itself is located on a created default broker. If *AssemblyContext2* and *AssemblyContext3* were both connected to a publish/subscribe channel with event group *A*, the resulting AMQP model would look as in Fig. 5.8. The publish/subscribe channel results in the two queues *A:2* and *A:3* which means that they are bound to the exchange with the routing key *A*. Hence, both queues receive the same messages.

At the moment, the messaging simulation does not offer an interface to a resource simulation. Therefore, information about the allocation of the entities is not translated. Moreover, the simulation does not support exchange to exchange or broker to broker bindings. Because the Palladio model representation was developed independent of RabbitMQ specifics, there are some constructs which can be modeled but can not be transformed to RabbitMQ entities: Components connected via a message channel are not transformed and the prefetch of connected receive endpoints is only translated if the connected receive endpoint belongs to an assembly context, i.e., results in a consumer, because RabbitMQ supports the setting of a prefetch only for consumers.

Reconfiguration	Implication on *IAmqpBrokerModel*
New assembly context encapsulating a messaging component	Creation of producers and consumers according to its connected send and receive endpoints and which channels they are connected to
Removal of assembly context encapsulating a messaging component	Deletion of producers and consumers corresponding to its connected send and receive endpoints

Table 5.3.: Overview of the anticipated messaging-related reconfigurations and their implication on the *IAmqpBrokerModel*

5.6. Simulating Self-Adaptations

Since we chose SimuLizar as a simulation engine for Palladio models, which is capable of simulating self-adaptive systems, most functionality needed for that is covered by it. For reasons of clarity, we present the process for defining self-adaptations of systems and how they are simulated with respect to the application for messaging-related self-adaptations.

The self-adaptations of a system can be specified by defining model-to-model transformations that will then be invoked every time a new measurement from a monitor that triggers self adaptations is available. Supported model transformation languages include QVT-O, Henshin, and Story Diagrams. In this work, we realized transformations using QVT-O (Queries, Views, and Transformations - Operational), an imperative transformation language using ImperativeOCL which is a variant of the Object Constraint Language (OCL) with side effects. Transformations must check if their respective precondition is fulfilled themselves. Typically, such a precondition is a measurement taken at a certain measuring point exceeding or falling below a defined threshold. We presented works about which metrics are suitable for scaling consumers of message queues were presented in Section 3.2. Transformations get the current Palladio model as input, modify that by e.g. adding or removing assembly contexts or resources, and output the modified Palladio model. SimuLizar then notifies all classes contributing an extension to the *modelobserver* extension point about the changes made.

The *MessagingModelSyncer* presented earlier in Section 5.3 contributes to that extension point and therefore gets notified about changes. It checks if a change concerns messaging-relevant parts of the system and adjusts the *IAmqpBrokerModel* according to it. The updated model is then forwarded to the *BrokerModelRuntimeState*.

Table 5.3 depicts possible reconfigurations related to messaging and their implications on the *IAmqpBrokerModel*. We derived the possible reconfigurations based on the anticipated replications of elements discussed in Section 4.2. The addition and removal of routers was not assumed to be a common scenario and is therefore not considered here.

While the model transformation is general, the transformation of changes in the model to the entities of the messaging simulation is specific to the messaging simulation used.

Transformations are always system-specific. However, we present an excerpt of a transformation in QVT-O for the example alarm notification system already used throughout Section 4.1 whose assembly is depicted in Fig. 4.7. Whenever a defined threshold for the

queue length of the *AlarmsFR* queue is exceeded and there are free resource containers, a new assembly context encapsulating the *Extractor* component is added which has its connected receive endpoints connected to the *AlarmsFR* Point-To-Point channel. Listing A.2 shows the method that checks for this condition and calls the `scaleUp` method if it is fulfilled. Listing A.1 depicts the `scaleUp` method of the transformation which is called to add the messaging assembly context and allocate it to a free resource container. *EncapsulatedComponentId*, *encapsulatedComponentAssemblyCtxId*, *replicationCount* are variables that are defined prior to calling the `scaleUp` function. The *replicationCount* variable simply counts how many times the transformation was executed to name the added assembly contexts accordingly. First, the instantiated component model object and the messaging assembly context to replicate are found by their ID. Then, a target resource container is chosen by selecting a random resource container out of all resource containers that are not allocated yet. In line 11, the new messaging assembly context is instantiated with the *object* keyword. It gets an ID and name affected by the current value of replication count to be uniquely identifiable. In the following, the connected receive endpoints of *assemblyContextToReplicate* are also copied to the *newAssemblyContext*. After that, the new assembly context is added to the messaging assembly as well as to the messaging allocation model. The functions `createConnectedReceiveEndpoints`, `addAssemblyContextToSystem`, `instantiateAllocationContext` are omitted. Fig. 5.9 shows the assembly model of the system after the transformation was executed. The added assembly context and its connected receive endpoint are colored orange.

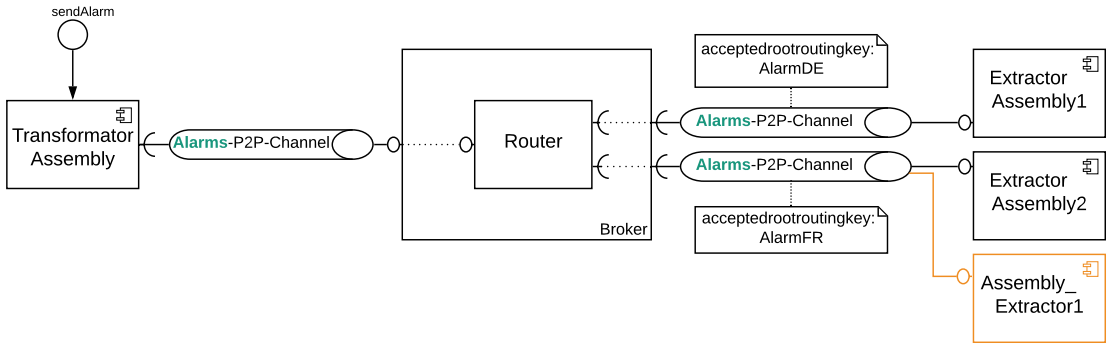

Figure 5.9.: Model after the transformation with added elements colored orange

After the transformation was executed, the *MessagingModelSyncer* gets notified about the change. It then checks if the change is messaging-relevant which is true since the new assembly context is a messaging assembly context. Hence, the *MessagingModelSyncer* must adapt the *IAmqpBrokerModel* to the new consumer by calling its `createConsumer(String consumerID, int prefetch, String routingEntity, String queueID)` method. This information can be derived from the model as we explained previously in Section 5.5. After this, he forwards the complete broker model to the *BrokerModelRuntimeState*. Fig. 5.10 depicts this process.

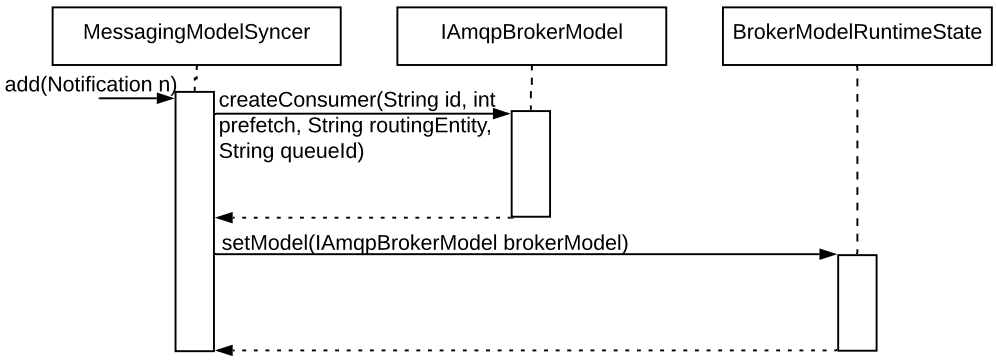

Figure 5.10.: Propagation of model changes by the *MessagingModelSyncer*

6. Evaluation

The main objective of this thesis is to enable design-time predictions for message-driven self-adaptive systems. Therefore, we introduced a meta-model for the representation of message-driven systems on an architectural level in Chapter 4. The simulation of message-driven self-adaptive systems based on SimuLizar and a messaging simulation was presented in Chapter 5. This chapter now evaluates whether the predictions made in the simulation comply with measurements taken during a real execution of the modeled system. Therefore, we investigate a case study: The system is executed and benchmarked during execution, and also modeled and simulated. The results are then compared regarding various metrics using a Goal-Question-Metric (GQM) plan [3] which is presented in further detail in Section 6.1. Section 6.2 is about the calibration of the messaging simulation. The case study used is introduced in Section 6.3. Section 6.4 presents the Palladio models for these scenarios. The results of the evaluation regarding prediction accuracy are described in Section 6.5 and regarding the support of self-adaptations in Section 6.6, followed by a discussion in Section 6.7.

6.1. GQM plan

The Goal-Question-Metric (GQM) method [3] is a top-down approach to ensure a structured and transparent procedure. First, goals are formulated. Then, questions are defined whose answers provide insights about whether the respective goal is achieved. The questions are answered using the defined metrics for that question.

The goals, questions, and metrics used for the evaluation of this thesis are shown in Fig. 6.1. They are oriented at the research questions formulated in Section 1.2. The overall goal is, as said earlier, to support design-time predictions for message-driven self-adaptive systems. It is divided into three sub-goals. First, the representation of message-driven systems on an architectural level should be possible. If the scenarios of the case study can be modeled, this goal is fulfilled. The second sub-goal is the increase of prediction accuracy for message-driven systems. The question here is two-fold: Can prediction accuracy for real systems be shown and if so, is the prediction more precise than predictions made using other approaches, e.g. performance completions? The metrics used are message latency, queue length, queue growth, distribution of messages to consumers, and memory consumption of the broker. The distribution to consumers refers to two levels here: First, if the routing and load balancing reflect the reality, and second if the flow control takes place. The prediction accuracy for these metrics will be evaluated by computing the average absolute and relative error, but also by assessing the qualitative correlation between measurements and prediction, e.g. if trends are foreseen correctly. The support of reconfigurations of message-driven systems is the third sub-goal. It is fulfilled if the

Goal 1	Support design-time predictions for message-driven self-adaptive systems
Goal 1.1	Enable representation of message-driven systems on architectural level
Question	Can case study scenarios be modeled using the proposed model elements?
Metric	Fulfilled: yes/no
Goal 1.2	Increase prediction accuracy for message-driven systems
Question	• Is the prediction more accurate than a prediction using performance completions? • Can the prediction accuracy be shown for real systems?
Metric	Latency of messages Queue length Queue growth Distribution of messages to consumers Memory consumption
Goal 1.3	Support reconfigurations of message-driven systems
Question	Can typical reconfigurations of message-driven systems be mapped to the architectural model and the messaging simulation?
Metric	Ratio of scenarios of the case study that can be mapped

Table 6.1.: Goals, Questions, and Metrics for the evaluation

reconfigurations of the system in the case study can be mapped to the architectural model and the messaging simulation.

The meta-model is not evaluated regarding comprehensibility and usability because it is difficult to avoid biases caused by the editor used for the model.

6.2. Calibrating the Model

We can not map all environmental factors to the model; mapping everything to the model is also not desirable because the model would lack its pragmatism then. Therefore, the model must be calibrated in order to obtain accurate performance predictions. Model calibration means enriching a qualitative model with quantitative data. This data can be estimated or collected from measurements [33, p. 116].

First, the benchmarks conducted for collecting the data are presented in Section 6.2.1. In Section 6.2.2, we explain the interpretation and usage of the obtained results for the calibration of the model.

6.2.1. Benchmarking RabbitMQ

Because the message size has a strong influence on the latency of a message and the performance of the broker [16, 17, 21], we are interested in the latency of messages of

OS	Amount of CPUs	CPU model	RAM
Ubuntu 18.04.4	48	AMD Opteron 6174	251 GB

Table 6.2.: Specifications of the machine used for benchmarking

different size for calibration of the model. The throughput, which is normally used for benchmarks, is not of interest here because it does not allow to draw conclusions about the latency of single messages and is therefore not useful for calibrating the model. The experiments measure the latency of messages always for one message size at a time. The *basic* benchmark configuration includes the sending of persistent messages via a direct exchange with one producer and one consumer with manual acknowledgments. The send rate of the producer is restricted to one, respectively hundred, messages per second to collect data for low and high load scenarios. The *pubAcks* configuration additionally includes required publisher acknowledgments, increasing the latency by the time for persisting the message, and the *prefetch* configuration a consumer prefetch of 1, increasing the latency by the time until consumer acknowledgments are received. This is expected to give insights about the time needed for message persisting and disposition sending.

We conducted the benchmarks with RabbitMQ PerfTest [29] version 2.11.0 on a machine from FZI with the specifications as shown in Tab. 6.2. RabbitMQ PerfTest, as well as RabbitMQ, ran in Docker containers connected by a bridge network. A docker image of RabbitMQ version 3.8.3 was used. The measured latency is end-to-end. Each test ran for one minute and was executed five times to account for possible inaccuracies of the measurements.

The results for the *basic* scenario with a send rate of 1 message per second are depicted in Fig. 6.1 and Fig. 6.2. Latencies with publisher acknowledgments can be seen in Fig. 6.5 and Fig. 6.6 latencies with a consumer prefetch of 1 in Fig. 6.3 and Fig. 6.4.

The latencies for a send rate of 100 msg/s are shown in Fig. 6.7 and Fig. 6.8 for the *basic* configuration, in Fig. 6.9 and Fig. 6.10 for the *prefetch* configuration and in Fig. 6.11 and Fig. 6.12 for the *pubAcks* configuration.

The average latency for messages in the base configuration and a send rate of 1 msg/s is higher than for a send rate of 100 msg/s. This could be caused by the send rate of 1 msg/s being too low to keep all RabbitMQ processes running and causing them to switch to an idle status in between the sending of two messages. Also, the *prefetch* configuration poses less overhead for a rate of 100 msg/s than for a rate of 1 msg/s. However, the *pubAcks* configuration is considerably slower for a rate of 100 msg/s.

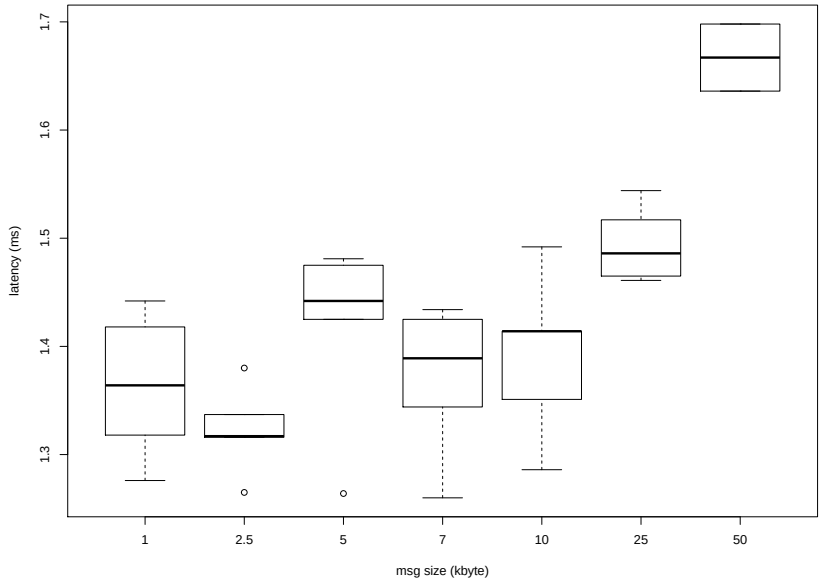

Figure 6.1.: Measured latency in the *base* configuration with a send rate of 1 msg/s

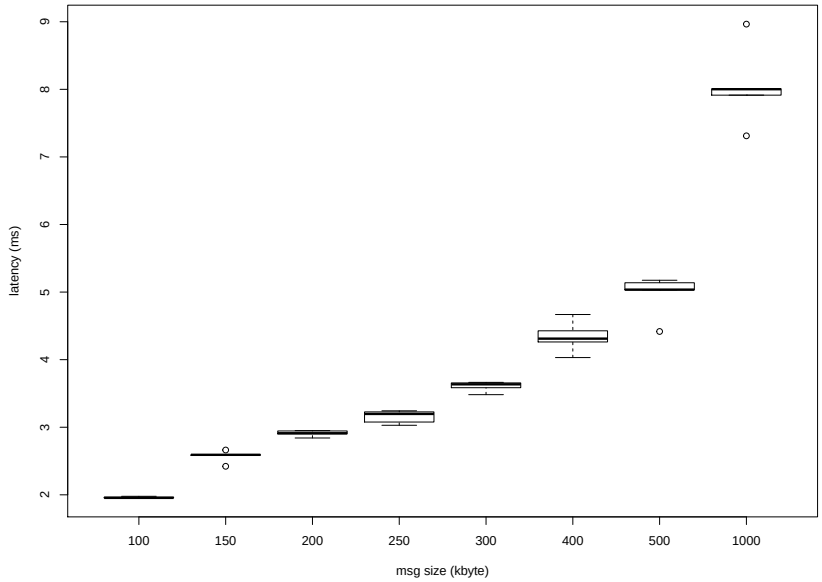

Figure 6.2.: Measured latency in the *base* configuration with a send rate of 1 msg/s

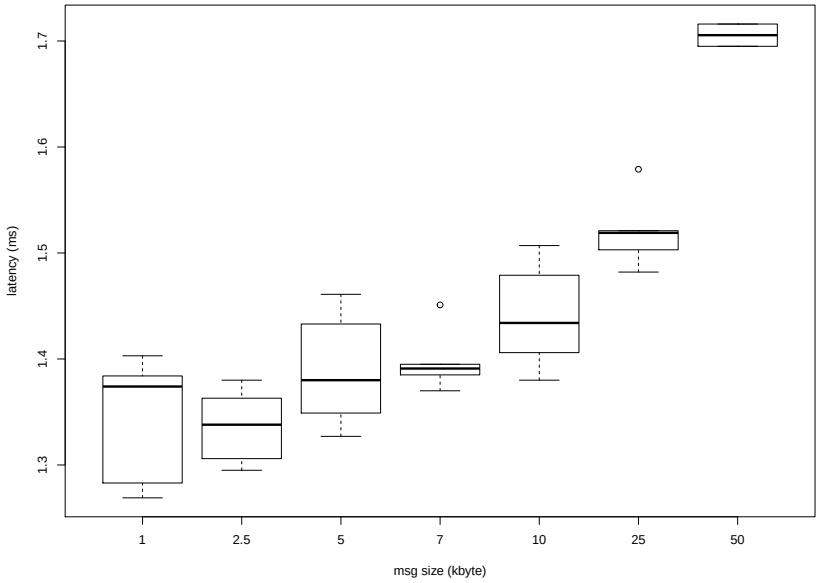

Figure 6.3.: Measured latency in the *prefetch* configuration with a send rate of 1 msg/s

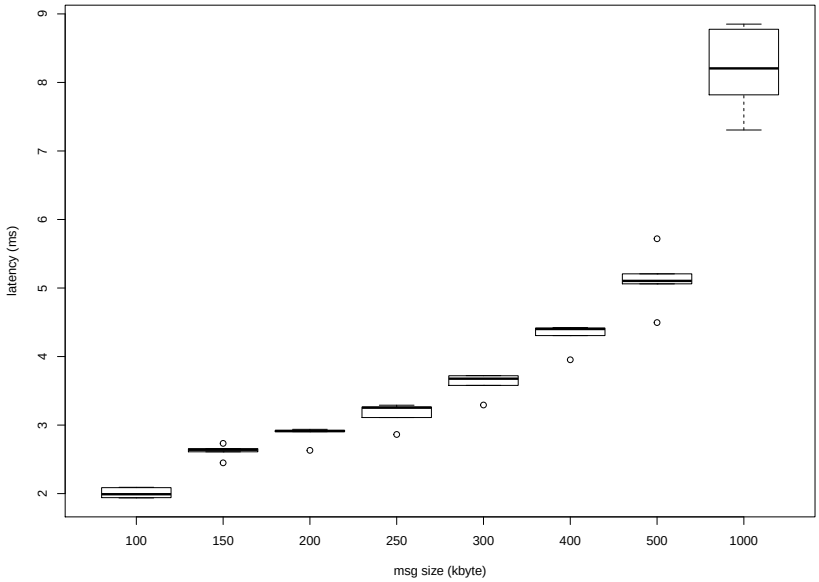

Figure 6.4.: Measured latency in the *prefetch* configuration with a send rate of 1 msg/s

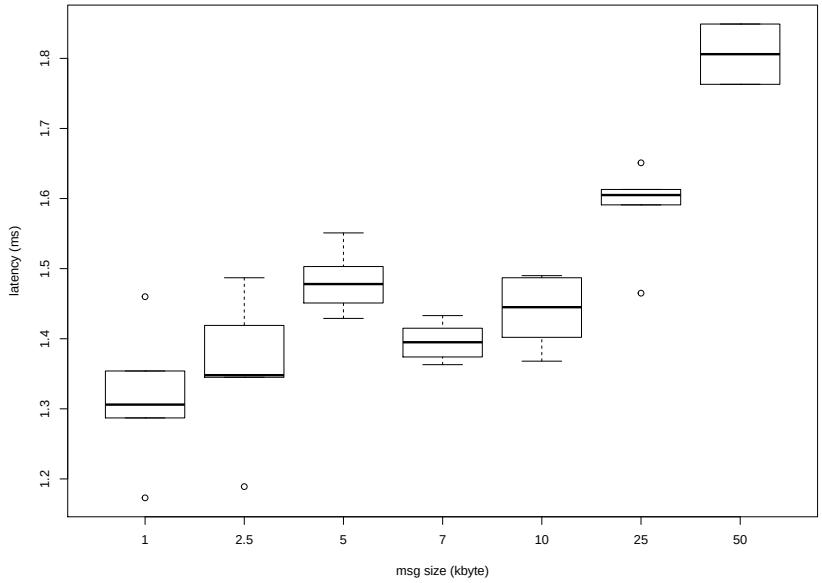

Figure 6.5.: Measured latency in the *pubAcks* configuration with a send rate of 1 msg/s

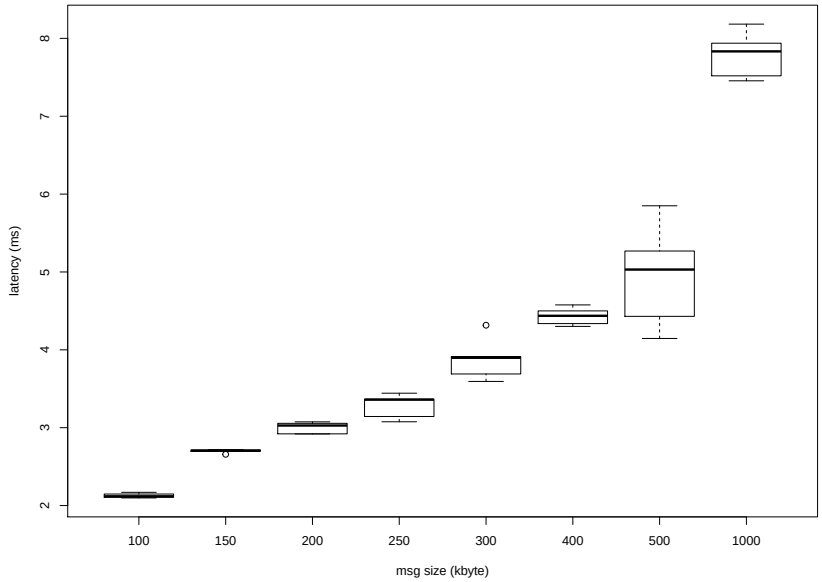

Figure 6.6.: Measured latency in the *pubAcks* configuration with a send rate of 1 msg/s

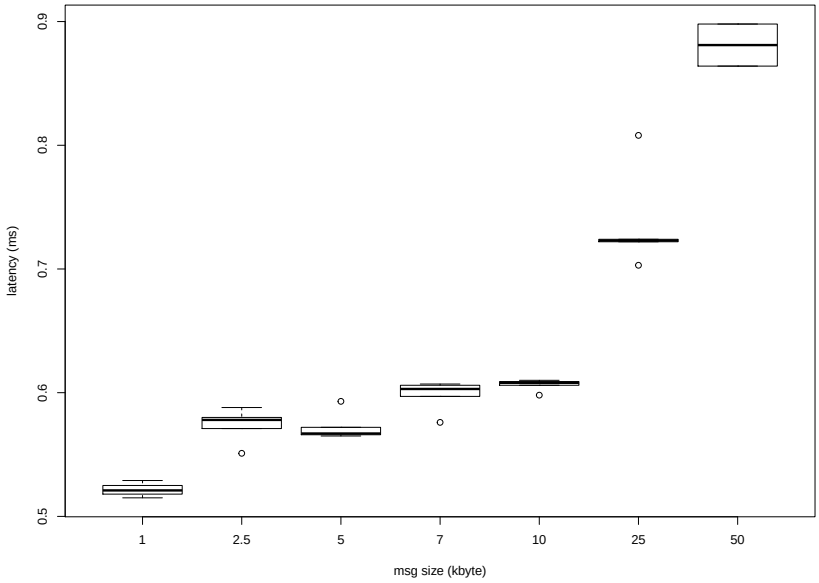

Figure 6.7.: Measured latency in the *base* configuration with a send rate of 100 msg/s

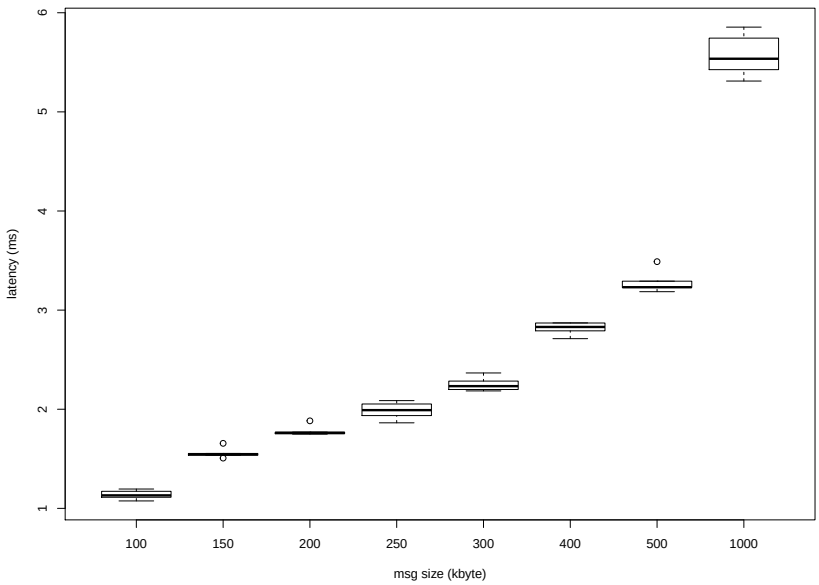

Figure 6.8.: Measured latency in the *base* configuration with a send rate of 100 msg/s

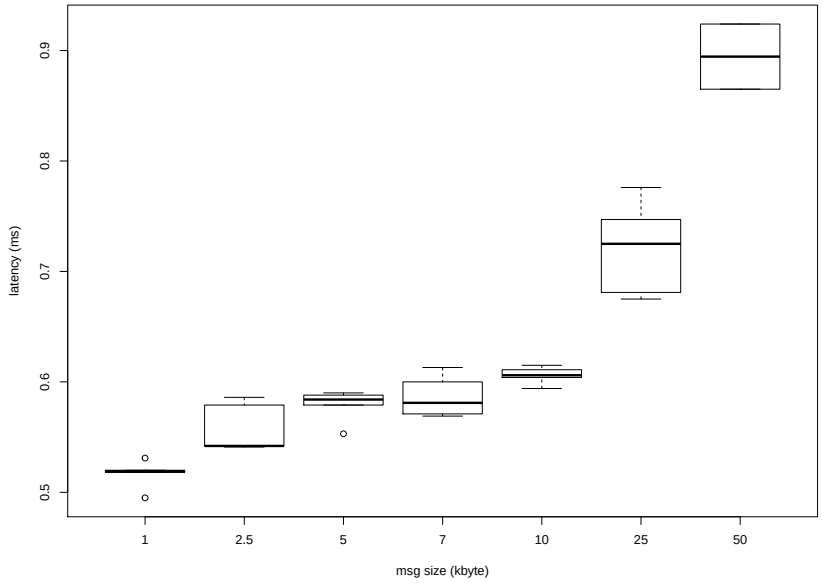

Figure 6.9.: Measured latency in the *prefetch* configuration with a send rate of 100 msg/s

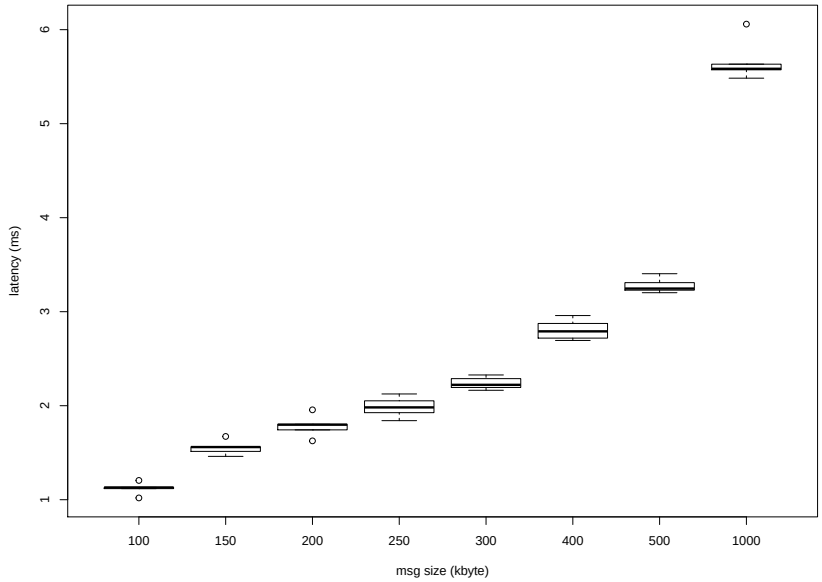

Figure 6.10.: Measured latency in the *prefetch* configuration with a send rate of 100 msg/s

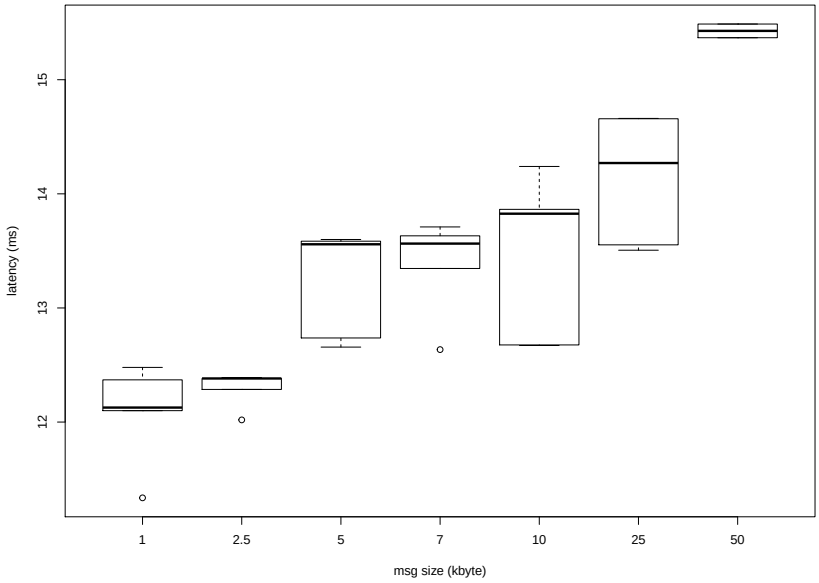

Figure 6.11.: Measured latency in the *pubAcks* configuration with a send rate of 100 msg/s

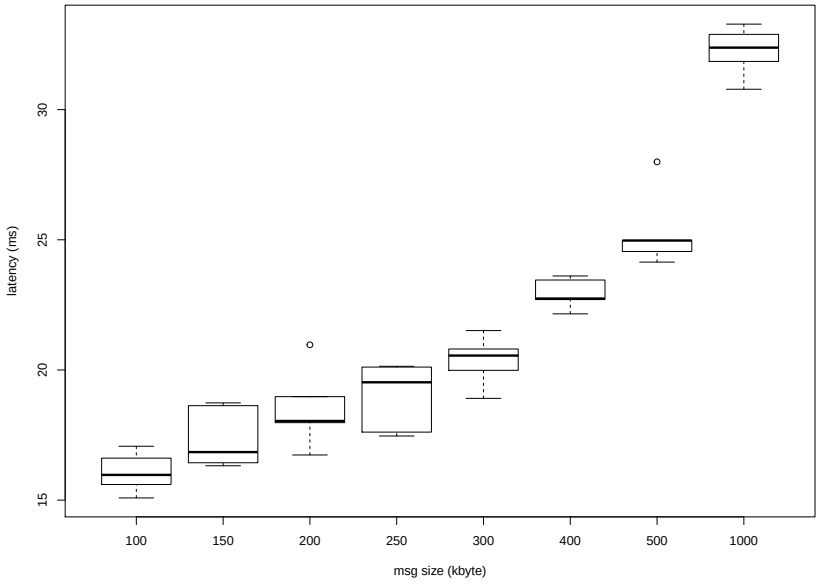

Figure 6.12.: Measured latency in the *pubAcks* configuration with a send rate of 100 msg/s

6.2.2. Interpreting the Results

The measurements conducted in Section 6.2.1 suggest a dependency between the size of a message and its latency. The Pearson Correlation Coefficient r normalizes the covariance of two random variables to values from minus 1 to plus 1 where a value of plus 1 suggests a strong positive correlation, minus 1 a strong negative correlation and a value of 0 no correlation. The p value describes the probability of observing a correlation of r even though the random variables are not correlated in the population. However, the Pearson Correlation Coefficient is suitable for linear dependencies only.

The result of calculating the Pearson Correlation Coefficient r is 0.99 for all of the measurements above with a p-value of 9.297897e-61 and smaller. This suggests a strong linear dependency between the size of a message and its latency with a low probability of wrong observation.

For estimating the latency of messages of arbitrary size, we used linear regression for interpolation. How well a linear regression fits the data can be measured by calculating the coefficient of determination r which provides insight into how much the data scatters along the linear regression. It ranges from 0 to 100% with higher values denoting smaller deviations. The regressions presented in the following achieve a r value of 98% to 99% except for the regression for the publisher acknowledgments with a r value of 95% which is not as high as for the other regressions but is still good.

The resulting linear regression model for the basic configuration with a send rate of 1 msg/s has the form given in Equation 1, for a send rate of 100 msg/s it has the form as in Equation 2. The respective plots are in Fig. 6.13 and in Fig. 6.16.

$$latency(ms) = 1.402349e-03 + (6.861179e-09 * messagesize(bytes)) \quad \text{(Equation 1)}$$

$$latency(ms) = 6.353401e-04 + (5.122246e-09 * messagesize(bytes)) \quad \text{(Equation 2)}$$

The overhead of the publisher acknowledgment and consumer prefetch features is determined by comparing the results of the respective measurements to the values of the basic scenario. For this reason, the values used for the respective linear regression are subtracted from each other and the resulting values are then used for calculating a new linear regression describing the overhead of the configuration.

For a send rate of 1 msg/s, the plots for the overhead introduced by the *prefetch* configuration can be seen in Fig. 6.14 and by the *pubAcks* configuration in Fig. 6.15. The overhead for the *pubacks* configurations decreases and even gets negative for higher message sizes. This does not allow for conclusions about the write rate of the message store that persists the messages. Therefore, only the regression for the overhead of the consumer prefetch, given in Equation 3, is used for calibrations, even though we did not expect the additional latency of the consumer prefetch to be dependent of the message size.

$$latency(ms) = 3.614767e-04 + (1.318373e-07 * messagesize(bytes)) \quad \text{(Equation 3)}$$

For a send rate of 100 msg/s, the overhead for the *prefetch* configuration is depicted in Fig. 6.17 and for the *pubAcks* configuration in Fig. 6.18. The overhead increases for the

pubAcks configuration now as expected and allows for conclusions about the time needed for persisting messages. The equation for the linear regression is given in Equation 4. Because Fig. 6.17 shows that messages sent in the *prefetch* configuration have an even shorter latency as in the *basic* scenario for sizes less than 200 KByte, the time needed for sending dispositions is assumed to be 0 in the high load scenario.

$$latency(ms) = 1.362617e-02 + (2.038807e-08 * messagesize(bytes)) \quad \text{(Equation 4)}$$

For calibrations of the model, we distinguish between high and low load scenarios for the base latency of a message which is set as in Eq. Equation 2 for high loads and as in Eq. Equation 1 for low loads. The write rate of the message store is set to be as in Eq. Equation 4. The time taken by dispositions sent by consumers is set to 0 even for the low load scenario since the additional latency there increases with message size and is therefore not assumed to be originated from the prefetch value but from the overhead of idle processes in between the sending of two messages.

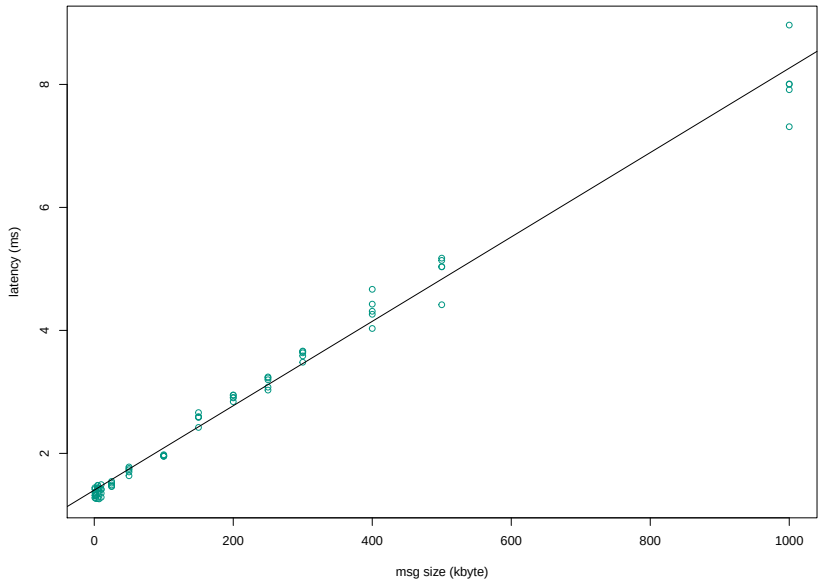

Figure 6.13.: Linear regression model for the basic configuration with a send rate of 1 msg/s

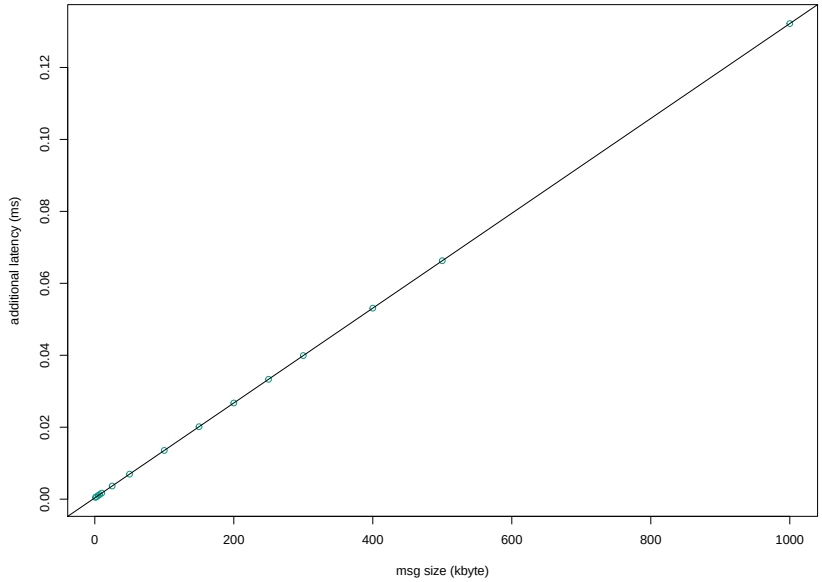

Figure 6.14.: Linear regression model for the difference between the configuration with consumer prefetch and the basic configuration for a send rate of 1 msg/s

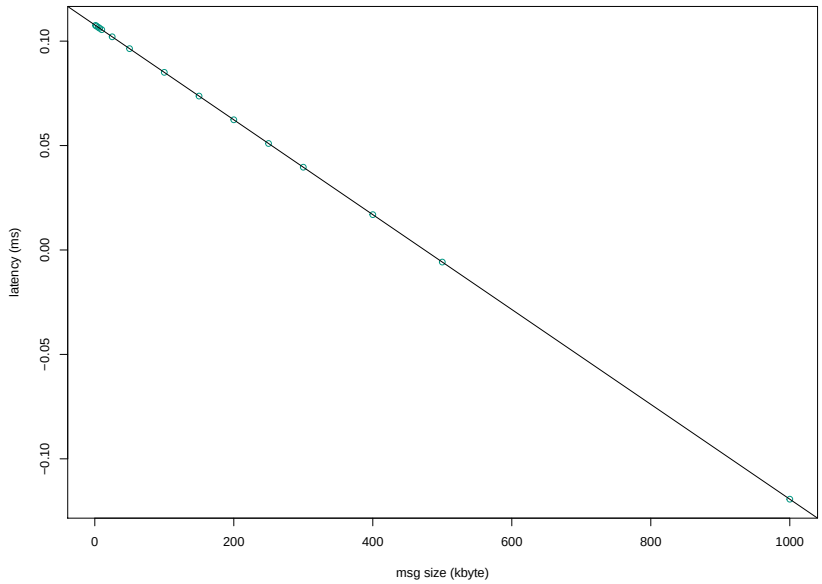

Figure 6.15.: Linear regression model for the difference between the configuration with publisher acknowledgments and the basic configuration for a send rate of 1 msg/s

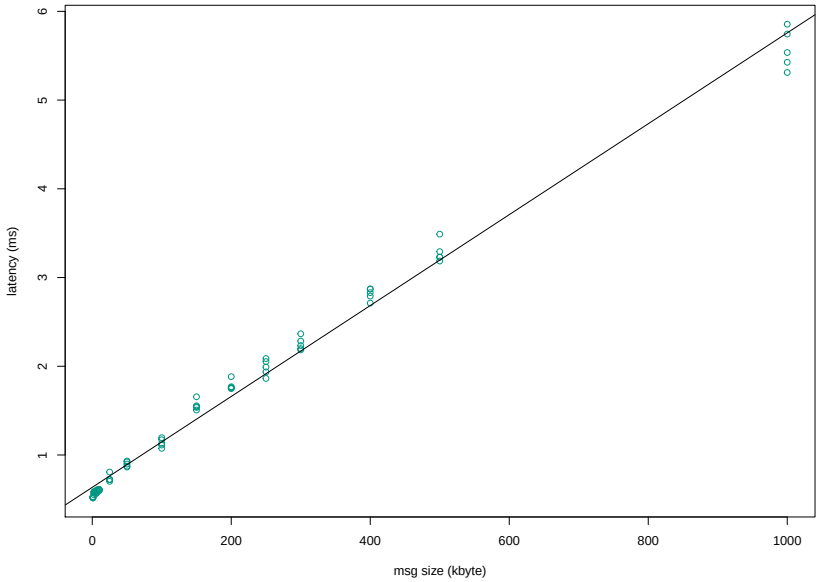

Figure 6.16.: Linear regression model for the basic configuration with a send rate of 100 msg/s

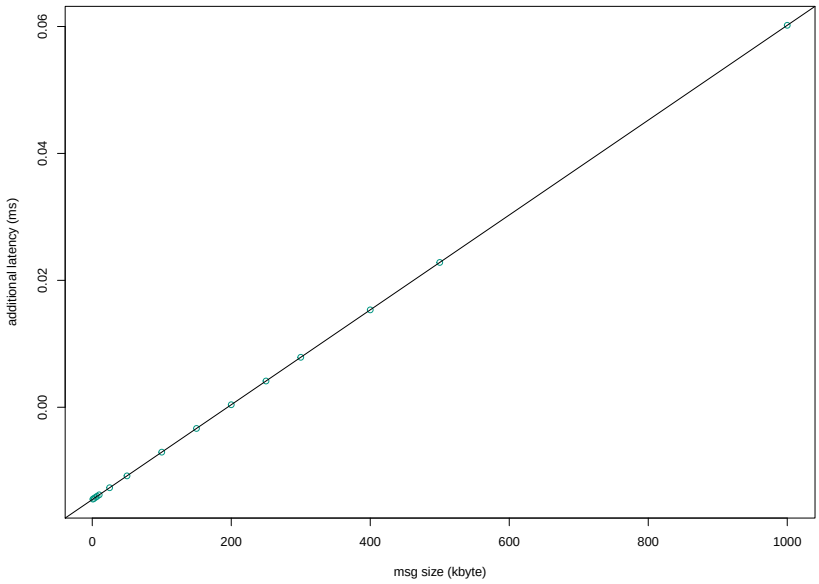

Figure 6.17.: Linear regression model for the difference between the configuration with consumer prefetch and the basic configuration for a send rate of 100 msg/s

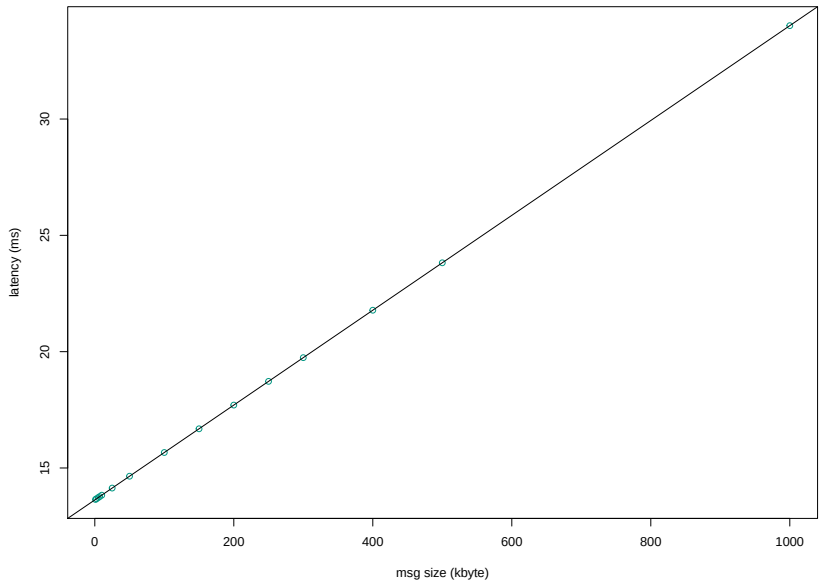

Figure 6.18.: Linear regression model for the difference between the configuration with publisher acknowledgments and the basic configuration for a send rate of 100 msg/s

6.3. SPECjms2007 Benchmark

The SPECjms2007 benchmark aims at providing a standard workload for measurement and evaluation of the performance and scalability of JMS-based MOMs [37]. It represents a real application and comprises seven different interactions, including both point-to-point and publish/subscribe communication. The messages sent have different types, sizes, and delivery modes. The system used in the benchmark models the supply chain of a supermarket company. There are four types of actors: Supplier (SP), Distribution Center (DC), Headquarter (HQ), and Supermarket (SM). Every supermarket is supplied by exactly one distribution center. When scaling horizontally, the number of supermarkets is changed whereas, for vertical scaling, the number of products sold is adjusted.

The scenario behavior of the actors is described in Section 6.3.1 and Section 6.3.2, where the former covers interaction 1 and the latter interaction 3. We chose the scenarios because they exercise Point-To-Point, respectively Publish/Subscribe communication. The other interactions are not considered for the evaluation in this work.

6.3.1. SPECjms2007 Interaction 1

The first interaction is about order and shipment handling and involves SMs, DCs, and the HQ. The configuration for this evaluation uses one SM, one DC, and one HQ. All communication is point-to-point and uses persistent messages. The sequence of events is as follows [37]:

Message type *Probability*	**Size 1** *95%*	**Size 2** *4%*	**Size 3** *1%*
Order	1.74	7.10	41.01
OrderConf	2.02	7.39	41.29
ShipDep	1.12	8.59	55.79
StatInfo	0.22	1.67	10.83
ShipInfo	1.28	8.76	55.95
ShipConf	0.81	2.73	14.83

Table 6.3.: Message sizes in KByte by message type for interaction 1 [34, p.117]

Message type *Probability*	**Size 1** *95%*	**Size 2** *4%*	**Size 3** *1%*
PriceUpdate	0.24	0.24	0.24

Table 6.4.: Message sizes of the *PriceUpdate* message in KByte for interaction 3 [34, p.117]

1. SM sends an order to its DC (*Order* message)
2. DC sends a confirmation to the SM (*OrderConf* message)
3. Goods are registered upon leaving the DC (*ShipDep* message)
4. DC sends information about the transaction to the HQ (*StatInfo* message)
5. Goods are registered at the SM upon arrival (*ShipInfo* message)
6. SM sends a confirmation to its DC (*ShipConf* message)

Tab. 6.3 depicts the sizes of messages in this interaction.

6.3.2. SPECjms2007 Interaction 3

This interaction describes the communication of price updates: When a selling price changes, the HQ sends *PriceUpdate* messages to the SMs. The configuration for this evaluation uses ten SMs and two HQs. Communication here is on a publish/subscribe basis and the messages sent are persistent. Tab. 6.4 depicts size and distribution of the *PriceUpdate* messages.

6.4. PCM Models of the SPECjms2007 Benchmark

For a performance prediction of the scenarios introduced in Section **??**, we modeled the scenarios as PCM models using the proposed model elements from Section 4.1. In the following, the PCM models for the SPECjms2007 interactions are described. The repository and system models are introduced in Section 6.4.1, Section 6.4.2 describes the Usage model. The resource environment and allocation model are omitted because the focus of the

Component	Sends messages	Receives messages
SM	*Order, ShipConf*	*OrderConf, ShipInfo*
DC	*OrderConf, StatInfo, ShipDep, ShipInfo*	*Order, ShipDep, ShipConf*
HQ		*StatInfo*

Table 6.5.: Sent and received mesage types by messaging component

evaluation is on metrics collected in the messaging simulation. Since the messaging simulation is not connected to a resource simulation, its metrics are independent of its allocation.

6.4.1. Repository and System model

The actors SM, DC, and HQ are modeled as messaging components. Every message sent in the two interactions is modeled as a message type within a separate event group. Each message type has a String input parameter *msg* which will be used to characterize the size of messages. There is no need for routing keys because the scenarios do not further distinguish messages apart from their type. The receive and send endpoints of the messaging components specify their respective event group. Tab. 6.5 provides an overview of which messaging component sends or receives which messages. The behavior upon receiving of a message is specified with RDSEFFs. In the case of interaction 1, receiving a message mostly triggers the sending of subsequent messages as described in Sec. 6.3.1 except for the *ShipConf* message which ends the interaction. In interaction 3, the SMs receiving the price updates do not execute further actions. Sending of messages is specified using the new *SendMessageAsync* action as it was introduced in Sec. 4.1.2. The size of the message to be sent, as presented in Tab. 6.3 and Tab. 6.4, is given with a *BYTESIZE* variable characterisation of the *msg* variable. To start interaction 1, the SM messaging component has an operation provided role providing an operation interface with a signature *sendOrder*, which sends an *Order* message. For the start of interaction 3, the HQ provides an interface with a *sendPriceUpdate* signature which sends a *PricdeUpdate* message.

Fig. 6.19 depicts the part of the system mapping interaction 1. All assembly contexts are connected to point-to-point channels via their connected send, respectively receive, endpoints. Every event group is sent via a different channel as depicted. The *contentbased* routing strategy of the router ensures that received messages of one event group are forwarded to an outgoing message channel of the same event group. Fig. 6.20 shows the part of the system mapping interaction 3. While both HQs send price updates via a point-to-point-Channel to the broker, the connected receive endpoints of the channels are connected to different connected receive endpoints of different routers via messaging delegation connectors. The connected send endpoints of both routers are connected to publish/subscribe channels. SM1 to SM5 are connected to the publish/subscribe channel Router1 publishes to, SM6 to SM10 to the publish/subscribe channel of Router2. This maps that one HQ is giving price updates to the five SMs it is responsible for.

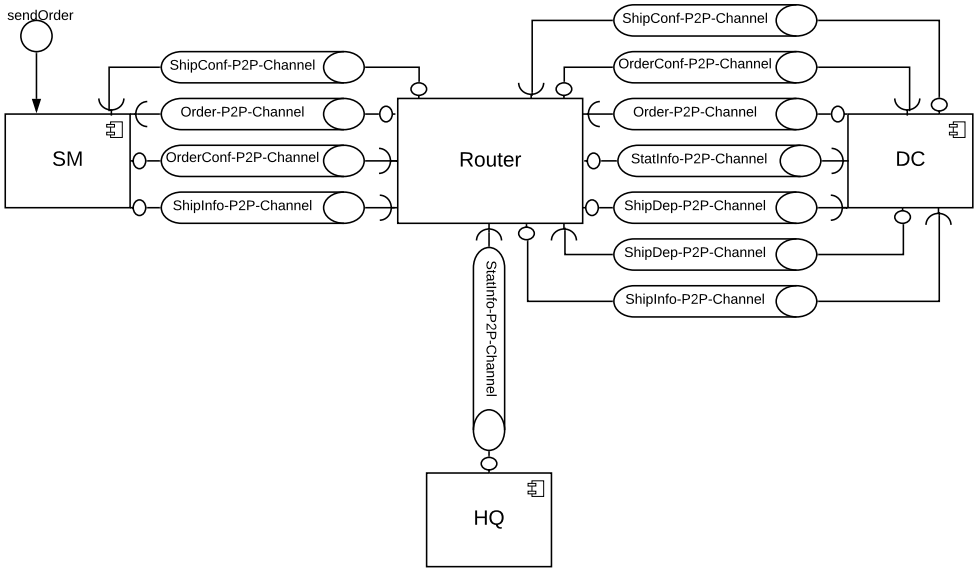

Figure 6.19.: System model of SPECjms2007 interaction 1

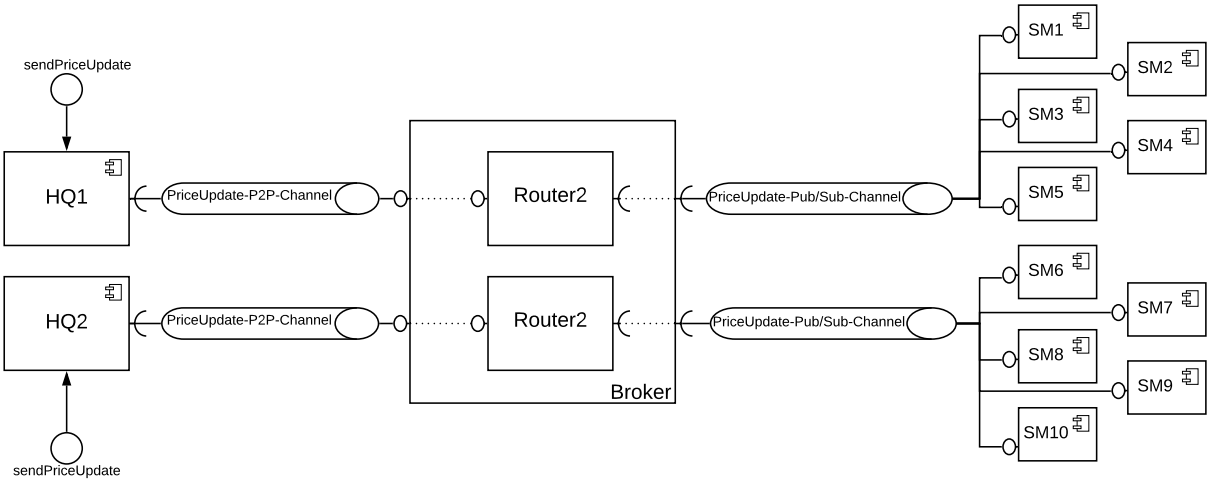

Figure 6.20.: System model of SPECjms2007 interaction 3

6.4.2. Usage model

The usage model contains a *UsageScenario* for each interaction. In the usage scenarios, an *EntryLevelSystemCall* is executed which refers to the respective operation provided role starting that interaction. For interaction 3, two *EntryLevelSystemCalls* are executed refering to one HQ each. The usage of separate usage scenarios enables the setting of an individual send rate for each interaction by using different workload specifications. Also, single interaction can be deactivated easily this way.

6.5. Evaluation of Prediction Accuracy

We evaluate the prediction accuracy by comparing the measurements obtained by running the SPECjms2007 benchmark against the values predicted by the simulation of a model of the SPECjms2007 scenarios as proposed in this thesis. The scenarios chosen are interaction 1 and 3 as they were introduced in Section 6.3. Interaction 1 ran with a send message rate of 1.539201540 msg/s and one SM, one DC, and one HQ. Interaction 3 ran with a send message rate of 6 msg/s and two HQs, each giving price updates to 5 SMs. This results in a send message rate of 3 msg/s per HQ. This does not adhere to the guidelines of SPEC which specify a different topology and the concurrent execution of all interactions.

We conducted each measurement five times for half an hour with a preceding warm-up period of 10 minutes in each case to account for possible inaccuracies of the measurements. They ran on the same machine as used for conducting the benchmarks in Section 6.2.1. Tab. 6.2 shows its specifications. The CPU utilization ranged from 5% to 30%. RabbitMQ as well as the SPECjms code was executed in docker containers communicating over a bridge network. The used version of RabbitMQ is 3.8.3.

As the respective send rates are low, the model calibration for low loads as presented in Section 6.2.2 is used. The CPU utilization is not measured because there is no actual processing of messages, actors only receive messages and forward them.

In addition to the evaluation of the case study scenarios, we have created several synthetic test cases to show the feasibility of the approach regarding flow control, scheduling, and use of different routing keys. They are shown in Tab. 6.6. The routing scenario includes one producer that sends messages with two different routing keys that have the same parent routing key to the broker. There are two consumer, each connected to a different point-to-point channel that each accepts one of the two routing keys. The outcome for all test cases is as expected.

6.5.1. Results SPECjms2007 Interaction 1

The measurement results for the latency of messages of interaction 1 of the SPECjms2007 benchmark and the performance prediction are presented in 6.21. The results from the five measurements conducted as well as the median of the measurements and the predicted latency are given by message type. The relative prediction error between the median of the measurements and the prediction in percent is between 2.75% and 36.87%, resulting in a median error of 13.21%. Tab. 6.7 depicts this. Also, the mean values of the measurements and the standard deviation are given. However, the median was used for calculating the

Test case	Tested property	Outcome
Over-producing producer, no transactional acknowledgments	Flow control	Messages are queued up and only delivered when the consumer has acknowledged the receipt of previous messages
Over-producing producer, transactional acknowledgments	Flow control	Messages are queued up and only delivered when the consumer has acknowledged previous messages after processing them
One producer, two consumers with same resources available, round-robin scheduling	Scheduling	Both consumers receive messages alternately
Routing scenario	Routing	Consumers only receive messages with the routing key of their channel

Table 6.6.: Synthetic test cases and their outcome

relative prediction error because it is not as strongly affected by outliers as the mean value. Tab. A.1 shows the measured latency per message type.

The predicted queue length of 0 is correct as well as the predicted queue growth of 0 and the rate of 1.53 of incoming messages which equals the rate at which *Order* messages are sent for all queues. The distribution to consumers is obviously correct as there is only one consumer connected to every queue. Flow control did not have any influence on deliveries, all entities sending messages had enough credit to send messages all the time. The prediction of memory consumption is qualitatively good as every message is held in RAM until it is acknowledged at a queue by the receiving consumer. Since every message gets consumed right away, this time interval is very short. However, the measurements did not show any RAM usage which is probably due to the time steps between the single measurements taken on the queue being too big and the time in RAM of a message being too short, which does not allow for a quantitative comparison.

6.5.2. Results SPECjms2007 Interaction 3

The median and average latency of the *PriceUpdate* message measured is 2.39ms. Tab. A.2 shows the measured values. The performance analysis predicts 1.403994ms, which gives us a prediction error of 41.42%. However, when the scenario is executed with only 1 SM instead of 10 SMs, the median latency of the *PriceUpdate* message is only 1.45ms with an average value of 1.48ms. The simulation still predicts a latency of 1.403994ms which results in a prediction error of only 3.17%. This implies that the higher latency is caused by the distribution of messages to the queues which has a higher resource demand when messages are forwarded to more queues. Because the messaging simulation is not yet connected to a resource simulation, this can not be reflected in the simulation at the moment.

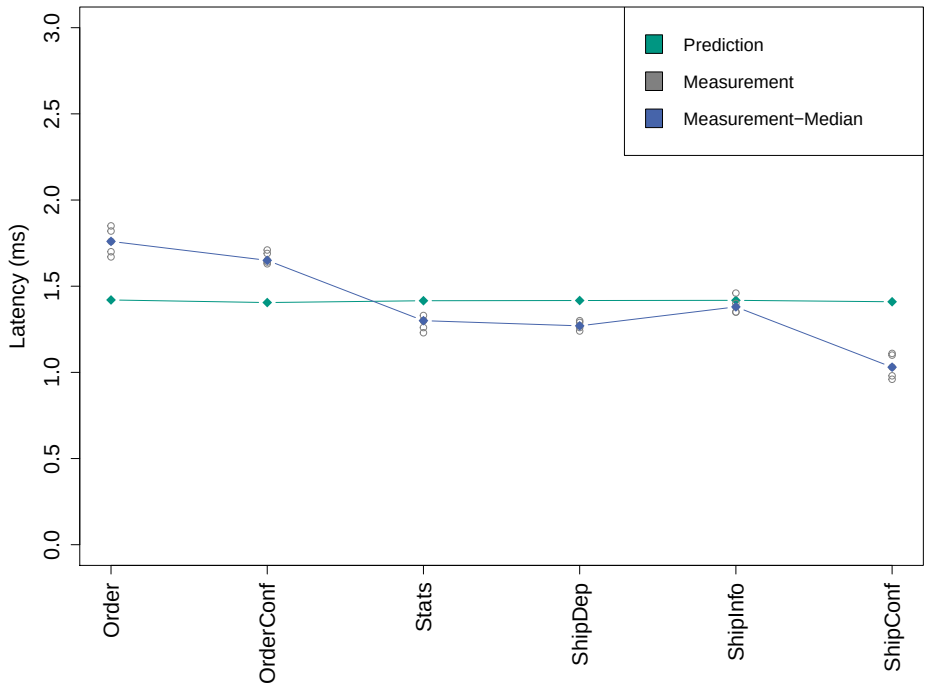

Figure 6.21.: Measurement results and prediction of latency for interaction 1 by message type

As before, the consumers consume the messages right after they are published to the queues, resulting in queue lengths of 0 which were predicted correctly. The queue growth of 0 is also correct for the whole measurement period. Every consumer receives every *PriceUpdate* sent which is in accordance with the measured values. Flow control had no impact on the deliveries as there was always enough credit available. Also, the memory consumption prediction shows the same behavior as before with every message being in RAM for a very short time. This is assumed to be correct because the messages are delivered almost immediately and are held in RAM until delivery only. However, as before, no quantitative comparison is possible due to the lack of measurements of RAM consumption.

6.6. Evaluation of Support of Self-Adaptations

The system in the case study is not self-adaptive. Therefore, we can not compare reconfigurations in the real system with predicted reconfigurations. However, we will argue that the approach in this work is capable of simulating self-adaptive message-driven systems that reconfigure based on message queue metrics and CPU utilization of consumers, as investigated by Gotin et al. [13].

In Section 4.2, we highlighted possible and anticipated replications of elements due to reconfigurations and presented how they can be mapped to the model. Section 5.6 covered which reconfigurations are expected during a simulation run. Tab. 6.8 shows

Message type	Order	OrderConf	Stats	ShipDep	ShipInfo	ShipConf
Median measurements (ms)	1.76	1.65	1.30	1.27	1.38	1.03
Average measurements (ms)	1.76	1.66	1.28	1.27	1.39	1.04
Standard deviation measurements (ms)	0.08	0.03	0.04	0.02	0.05	0.07
Prediction (ms)	1.420202	1.405133	1.415679	1.416923	1.417947	1.409733
Deviation between median measurement and prediction	19.31%	14.84%	8.90%	11.57%	2.75%	36.87%

Table 6.7.: Delivery time by message type and deviation between measurement medians and prediction

considered reconfigurations and their implications on the Palladio model, as well as their implications on the AMQP simulation model. We explained how self-adapting behavior can be defined and which changes in the model are propagated how to the messaging simulation model. This was illustrated by an example adaptation that adds more consumers to a queue if the length of the queue exceeds a defined threshold. Therefore, we presented a transformation that instantiates additional message receiving messaging assembly contexts to that channel. We also described the process of propagating the new assembly context, respectively consumer, to the messaging simulation model.

As mentioned, the execution of transformations depends on a precondition they specify. Typically, this condition checks if the value of a measurement at a measuring point exceeds or is below a defined threshold. Since we added measuring points to collect measurements at the new model elements as explained in Section 5.4, carrying out transformations dependent on messaging metrics is possible. In fact, all metrics investigated by Gotin et al. [13] which were presented in Section 3.2 can be used.

To show the feasibility of the approach, we have created synthetic test cases that added and removed consumers based on the queue length and the response time of consumers.

Gotin et al. [14] also mentioned the possibility to adapt send rates to provide overload protection. Though overload protection can be provided by the consumer prefetch and the flow control by the broker and consumers on the link layer, an adaptation of the send rate would also be possible by applying a transformation to the workload of the respective usage scenario.

SimuLizar also supports transient effect analysis [40] by specifying the adaptation behavior in an action repository that can cause resource demands. Transient effects caused by link creation and removal are already considered by the messaging simulation. However, not all transient effects can be mapped since the messaging simulation is not yet connected to a resource simulation.

Reconfiguration	Implication on Palladio model	Implication on AMQP broker model
Adding a new consumer	New assembly context that receives messages	Creation of consumers according to connected receive endpoints
Removal of consumer	Removal of assembly context that receives messages	Removal of corresponding consumer
Adding a new producer	New assembly context that sends messages	Creation of producers according to connected send endpoints
Removal of producer	Removal of assembly context that sends messages	Removal of corresponding producer

Table 6.8.: Overview of the anticipated messaging-related reconfigurations and their implication on the Palladio model and the *IAmqpBrokerModel*

To sum up, we provided concepts on how reconfigurations of a message-driven system can be mapped to the model and propagated further to the entities of the messaging simulation used. The addition and removal of messaging assembly contexts were realized on an exemplary basis by QVT-O transformations. We, therefore, expect to be able to map typical reconfigurations of a message-driven system to the architectural model as well as to the messaging simulation. However, the performance predictions may be over-optimistic since we did not consider the cost of reconfigurations yet.

6.7. Evaluation summary

To evaluate the approach proposed in this thesis, we defined three evaluation goals that focus on the applicability of the model elements, the prediction accuracy, and the support for reconfigurations of the system, as depicted in Tab. 6.1.

We evaluated these goals using the SPECjms2007 case study. Interaction 1 and 3 of it were modeled using the model elements proposed in Section 4.1 which showed their applicability. Both interactions could be modeled in full detail. The comparison of predicted and measured message latency exhibited a median prediction error of 13.21% for Point-To-Point communication and a prediction error of 41.42% for Publish/Subscribe communication. Modeling and simulating message-driven systems as proposed by Rathfelder [31] exhibited a prediction error not exceeding 25% in most of the cases which means that the results obtained by using the approach from this thesis could be slightly better without measuring resource demands separately for every message but using a generic calibration only. However, the prediction accuracy for Publish/Subscribe communication could not be shown as the prediction error is considered too high. This could be improved by connecting the messaging simulation to a resource simulation as implied by the measurements taken when messages were delivered to one SM instead of ten. Besides that, queue length, as well as queue growth and input rates, were predicted correctly for both interactions of the SPECjms2007 case study. The memory consumption is qualitatively correct. This shows

that more detailed statements about the state of the system are possible using a dedicated messaging simulation. However, for making more reliable statements about the prediction accuracy of this approach, a larger case study needs to be investigated.

An internal threat to the validity of the evaluation is the small sample. We chose to evaluate only two interactions which are not necessarily representative for message-driven systems. Furthermore, the messaging simulation is still under development and could still include wrong behavior and bugs. External validity is threatened by the fact that case studies can not be generalized. Consequently, showing the prediction accuracy for one case study does not allow for conclusions about the prediction accuracy in other scenarios.

7. Conclusion

In this last chapter, we conclude the thesis and give an overview of possibilities for future work.

7.1. Summary

The goal of this thesis has been to enable quality predictions for message-driven self-adaptive systems. Previous approaches did not simulate messaging in detail and abstracted queuing behavior among other factors. Moreover, they did not consider self-adaptive systems. Therefore, this thesis presented a meta-model for architectural representation of messaging, and a simulation interface between a simulation of a component-based architecture description language and a messaging simulation. We introduced an implementation of this interface for the Palladio simulation engine SimuLizar and an AMQP messaging simulation, and explained how self-adaptations can be specified and simulated. The approach has been evaluated regarding its applicability, prediction accuracy, and support of reconfigurations of self-adaptive systems.

We developed the meta-model as an extension to the PCM following the patterns of Hohpe and Woolfs [18]. This ensures the independence of the meta-model from message broker implementations. It comprises four different view types, separating the modeling of structural, behavioral, and deployment characteristics, as well as system-specific from system-independent characteristics. Colored Petri Nets have been applied in order to define the semantics of the model elements. We also highlighted expected reconfigurations of systems and their respective mapping to models of the meta-model.

To analyze the impact of using messaging on component-based systems, we defined a number of requirements for a simulation interface between a simulation of a component-based architecture description language and a messaging simulation, and presented a generic concept for such an interface. With regard to the implementation, we chose the Palladio simulation engine SimuLizar and used an AMQP messaging simulation. We chose SimuLizar since systems using messaging tend to be dynamic and SimuLizar is able to simulate self-adaptations. This realization integrates two models within one simulation to ensure consistency and synchronization between both of them on the technical side. The use of a dedicated messaging simulation enables the prediction of other quality-related metrics than message latency, such as length of message queues; it also facilitates the distribution of messages to consumers not only for reasons of load balancing but also conditional on a flow control. This provides deeper insights into the quality of a system.

In the evaluation chapter, we demonstrated the applicability of the proposed model elements by modeling a representative case study. Regarding the latency of messages, we could show the prediction accuracy for Point-To-Point communication. For Publish/Sub-

scribe communication, the prediction error was 41.42 percent, which means that prediction accuracy could not be shown for this communication model. Other metrics, such as queue length and input rate, proved to be accurate for Point-To-Point and Publish/Subscribe communication. The RAM consumption of queues was qualitatively correct. We also argued that the approach is capable of supporting the reconfiguration of systems. However, further case studies have to be performed to verify the general applicability of the approach and its prediction accuracy.

7.2. Future Work

Future studies could benefit from further investigation of several aspects. At the moment, the messaging simulation is not connected to a resource simulation. The evaluation of the Publish/Subscribe scenario showed that the latency of messages increases as more consumers subscribe to a specific Publish/Subscribe channel. By connecting a resource simulation to the messaging simulation, this behavior could be mapped and prediction accuracy therefore be improved. Prediction accuracy could be improved further through the use of a theoretical model describing the time certain events take in more detail. On the implementation side, modeling could be eased by providing graphical editors that extend existing Palladio editors.

Bibliography

[1] *Abstract Simulation Engine*. https://sdqweb.ipd.kit.edu/wiki/Abstract_Simulation_Engine. Last accessed: 2020-05-02.

[2] *ArchiMate*. https://www.opengroup.org/archimate-forum/archimate-overview. Last accessed: 2019-11-14.

[3] V. R. Basili and D. M. Weiss. "A Methodology for Collecting Valid Software Engineering Data". In: *IEEE Transactions on Software Engineering* SE-10.6 (1984), pp. 728–738.

[4] Matthias Becker, Steffen Becker, and Joachim Meyer. "SimuLizar: Design-Time Modeling and Performance Analysis of Self-Adaptive Systems". In: *Software Engineering 2013*. Ed. by Stefan Kowalewski and Bernhard Rumpe. Bonn: Gesellschaft für Informatik e.V., 2013, pp. 71–84.

[5] Matthias Becker, Markus Luckey, and Steffen Becker. "Performance Analysis of Self-adaptive Systems for Requirements Validation at Design-time". In: *Proceedings of the 9th International ACM Sigsoft Conference on Quality of Software Architectures*. QoSA '13. Vancouver, British Columbia, Canada: ACM, 2013, pp. 43–52. ISBN: 978-1-4503-2126-6. DOI: 10.1145/2465478.2465489. URL: http://doi.acm.org/10.1145/2465478.2465489.

[6] Stipe Celar, Eugen Mudnic, and Zeljko Seremet. "State-Of-The-Art of Messaging for Distributed Computing Systems". In: Jan. 2016, pp. 0298–0307. ISBN: 9783902734082. DOI: 10.2507/27th.daaam.proceedings.044.

[7] Betty H.C. Cheng et al. "08031 – Software Engineering for Self-Adaptive Systems: A Research Road Map". In: *Software Engineering for Self-Adaptive Systems*. Ed. by Betty H. C. Cheng et al. Dagstuhl Seminar Proceedings 08031. Dagstuhl, Germany: Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, Germany, 2008. URL: http://drops.dagstuhl.de/opus/volltexte/2008/1500.

[8] *Communication Path*. https://circle.visual-paradigm.com/communication-path/. Last accessed: 2019-11-14.

[9] Edward Curry. "Message-Oriented Middleware". In: *Middleware for Communications*. John Wiley & Sons, Ltd, 2005. Chap. 1, pp. 1–28. ISBN: 9780470862087. DOI: 10.1002/0470862084.ch1. eprint: https://onlinelibrary.wiley.com/doi/pdf/10.1002/0470862084.ch1. URL: https://onlinelibrary.wiley.com/doi/abs/10.1002/0470862084.ch1.

[10] Thomas Czogalik. "Modellierung und Simulation von verteilter und wiederverwendbarer nachrichtenbasierter Middleware". MA thesis. Karlsruhe Institute of Technology (KIT), 2019.

[11] *Display messages in queues*. https://wiki.vianovaarchitectura.nl/index.php/Display_messages_in_queues. Last accessed: 2019-11-14.

[12] *Eclipse Modeling Framework*. https://www.eclipse.org/modeling/emf/.

[13] Manuel Gotin et al. "Investigating Performance Metrics for Scaling Microservices in CloudIoT-Environments". In: *ICPE*. 2018.

[14] Manuel Gotin et al. "Overload Protection of Cloud-IoT Applications by Feedback Control of Smart Devices". In: *Proceedings of the 2019 ACM/SPEC International Conference on Performance Engineering*. ICPE '19. Mumbai, India: Association for Computing Machinery, 2019, pp. 51–58. ISBN: 9781450362399. DOI: 10.1145/3297663.3309673. URL: https://doi.org/10.1145/3297663.3309673.

[15] AMQP Working Group. *AMQP - Advanced Message Queuing Protocol Protocol Specification*. Nov. 2009. URL: http://www.amqp.org/specification/0-9-1/amqp-org-download.

[16] Jens Happe et al. "A Pattern-Based Performance Completion for Message-Oriented Middleware". In: *Proceedings of the 7th International Workshop on Software and Performance*. WOSP '08. Princeton, NJ, USA: Association for Computing Machinery, 2008, pp. 165–176. ISBN: 9781595938732. DOI: 10.1145/1383559.1383581. URL: https://doi.org/10.1145/1383559.1383581.

[17] Jens Happe et al. "Parametric Performance Completions for Model-Driven Performance Prediction". In: *Performance Evaluation* 67.8 (2010), pp. 694–716.

[18] Gregor Hohpe and Bobby Woolf. *Enterprise integration patterns : designing, building, and deploying messaging solutions*. 10. print. The Addison-Wesley signature series. Boston, Mass. [u.a.]: Addison-Wesley, 2007. ISBN: 0321200683; 9780321200686.

[19] "IEEE Standard for Modeling and Simulation (M&S) High Level Architecture (HLA)–Framework and Rules". In: *IEEE Std 1516-2010 (Revision of IEEE Std 1516-2000)* (2010), pp. 1–38.

[20] *Indirections Git Repository*. https://github.com/PalladioSimulator/Palladio-Addons-Indirections.

[21] V. M. Ionescu. "The analysis of the performance of RabbitMQ and ActiveMQ". In: *2015 14th RoEduNet International Conference - Networking in Education and Research (RoEduNet NER)*. Sept. 2015, pp. 132–137. DOI: 10.1109/RoEduNet.2015.7311982.

[22] Kurt Jensen. "Coloured petri nets and the invariant-method". In: *Theoretical Computer Science* 14.3 (1981), pp. 317–336. ISSN: 0304-3975. DOI: https://doi.org/10.1016/0304-3975(81)90049-9. URL: http://www.sciencedirect.com/science/article/pii/0304397581900499.

[23] Kurt Jensen and Lars M. Kristensen. *Coloured Petri Nets: Modelling and Validation of Concurrent Systems*. 1st. Springer, Berlin, Heidelberg, 2009. ISBN: 978-3-642-00284-7. DOI: https://doi.org/10.1007/b95112.

[24] Floriment Klinaku, Dominik Bilgery, and Steffen Becker. “The Applicability of Palladio for Assessing the Quality of Cloud-based Microservice Architectures”. In: *Proceedings of the 13th European Conference on Software Architecture - Volume 2*. ECSA ’19. Paris, France: ACM, 2019, pp. 34–37. ISBN: 978-1-4503-7142-1. DOI: 10.1145/3344948.3344961. URL: http://doi.acm.org/10.1145/3344948.3344961.

[25] Sebastian Lehrig. *Quality Analysis Lab (QuAL): Software Design Description and Developer Guide*. Tech. rep. Universität Paderborn, 2016.

[26] Tania Lorido-Botran, Jose Miguel-Alonso, and Jose A Lozano. “A review of auto-scaling techniques for elastic applications in cloud environments”. In: *Journal of grid computing* 12.4 (2014), pp. 559–592.

[27] *Messaging Simulation*. https://git.rss.iste.uni-stuttgart.de/mosaic/mosaic-broker.

[28] OASIS. *OASIS Advanced Message Queuing Protocol (AMQP) Version 1.0*. Oct. 2012. URL: https://docs.oasis-open.org/amqp/core/v1.0/amqp-core-complete-v1.0.pdf.

[29] *RabbitMQ PerfTest*. https://rabbitmq.github.io/rabbitmq-perf-test/stable/htmlsingle/. Last accessed: 2020-04-25.

[30] *RabbitMQ website*. https://www.rabbitmq.com/. Last accessed: 2020-05-02.

[31] C. Rathfelder. *Modelling Event-Based Interactions in Component-Based Architectures for Quantitative System Evaluation*. Karlsruhe Series on Software Design and Quality / Ed. by Prof. Dr. Ralf Reussner. KIT Scientific Publishing, 2013. ISBN: 9783866449695. URL: https://books.google.de/books?id=BbYKjU4j2cAC.

[32] Christoph Rathfelder et al. “Modeling event-based communication in component-based software architectures for performance predictions”. In: *Software & Systems Modeling* 13.4 (Oct. 2014), pp. 1291–1317. ISSN: 1619-1374. DOI: 10.1007/s10270-013-0316-x. URL: https://doi.org/10.1007/s10270-013-0316-x.

[33] Ralf [HerausgeberIn] Reussner et al., eds. *Modeling and simulating software architectures : the Palladio approach*. Includes bibliographical references and index. Cambridge, Massachusetts: The MIT Press, [2016]. ISBN: 9780262034760. URL: http://www.gbv.de/dms/tib-ub-hannover/85684540x.pdf%20;%20https://mitpress.mit.edu/modeling.

[34] Kai Sachs. “Performance Modeling and Benchmarking of Event-Based Systems”. PhD thesis. TU Darmstadt, 2011.

[35] Kai Sachs, Samuel Kounev, and Alejandro Buchmann. “Performance modeling and analysis of message-oriented event-driven systems”. In: *Software & Systems Modeling* 12.4 (Oct. 2013), pp. 705–729. ISSN: 1619-1374. DOI: 10.1007/s10270-012-0228-1. URL: https://doi.org/10.1007/s10270-012-0228-1.

[36] Douglas C Schmidt. “Guest Editor’s Introduction: Model-Driven Engineering”. In: *Computer - IEEE Computer Society* - 39.2 (Feb. 2006), pp. 25–31. DOI: 10.1109/MC.2006.58.

[37] *SPECjms2007 Design Document*. https://www.spec.org/jms2007/docs/DesignDocument.html. Last accessed: 2020-04-24.

[38] Herbert Stachowiak. *Allgemeine Modelltheorie*. Springer-Verlag Wien-New York, 1973.

[39] Thomas Stahl and Markus Völter. *Model driven software development : technology, engineering, management*. Chichester [u.a.]: Wiley, 2006. ISBN: 0-470-02570-0; 978-0-470-02570-3.

[40] C. Stier and A. Koziolek. "Considering Transient Effects of Self-Adaptations in Model-Driven Performance Analyses". In: *2016 12th International ACM SIGSOFT Conference on Quality of Software Architectures (QoSA)*. Venice, 2016, pp. 80–89. DOI: 10.1109/QoSA.2016.14.

[41] A. Tolk. "Interoperability, Composability, and Their Implications for Distributed Simulation: Towards Mathematical Foundations of Simulation Interoperability". In: *2013 IEEE/ACM 17th International Symposium on Distributed Simulation and Real Time Applications*. 2013, pp. 3–9.

[42] Andreas Tolk and James A Muguira. "The levels of conceptual interoperability model". In: *Proceedings of the 2003 fall simulation interoperability workshop*. Vol. 7. 2003, pp. 1–11.

[43] Antti Valmari. "The state explosion problem". In: *Lectures on Petri Nets I: Basic Models: Advances in Petri Nets*. Ed. by Wolfgang Reisig and Grzegorz Rozenberg. Berlin, Heidelberg: Springer Berlin Heidelberg, 1998, pp. 429–528. ISBN: 978-3-540-49442-3. DOI: 10.1007/3-540-65306-6_21. URL: https://doi.org/10.1007/3-540-65306-6_21.

[44] Wenguang Wang, Andreas Tolk, and Weiping Wang. "The Levels of Conceptual Interoperability Model: Applying Systems Engineering Principles to M & S". In: *Proceedings of the 2009 Spring Simulation Multiconference*. SpringSim '09. San Diego, California: Society for Computer Simulation International, 2009.

A. Appendix

A.1. QVT-O Transformations

```
1   helper scaleUp(allocation : MessagingAllocation) : Boolean {
2   var instantiatedComponent : BasicComponent := allocation.system_Allocation.
        assemblyContexts__ComposedStructure.encapsulatedComponent__AssemblyContext[
        BasicComponent]
3   ->any(id = encapsulatedComponentId);
4   var assemblyContextToReplicate : MessagingAssemblyContext := allocation.
        allocationContexts_Allocation.assemblyContext_AllocationContext->select(id =
        encapsulatedComponentAssemblyCtxId)->oclAsType(MessagingAssemblyContext)->any(true);
5
6   var targetContainers := allocation.targetResourceEnvironment_Allocation.
        resourceContainer_ResourceEnvironment
7   ->reject(r | allocation.allocationContexts_Allocation.resourceContainer_AllocationContext
        ->includes(r));
8   var targetResourceContainer : ResourceContainer := targetContainers->any(true);
9
10  //create new assemblyContext.
11  var newAssemblyContext = object MessagingAssemblyContext {
12  entityName := 'Assembly_' + instantiatedComponent.entityName + replicationCount.toString()
        ;
13  id := assemblyContextToReplicate.id + replicationCount.toString();
14  encapsulatedComponent__AssemblyContext := instantiatedComponent;
15  };
16  newAssemblyContext.connectedendpoint := createConnectedReceiveEndpoints(
        assemblyContextToReplicate.connectedendpoint);
17
18  //add assemblyContext to messaging assembly
19  msgAssembly.rootObjects()[MessagingAssembly]->map addAssemblyContextToSystem(
        newAssemblyContext);
20  msgAllocation.rootObjects()[MessagingAllocation]->map instantiateAllocationContext(
        newAssemblyContext, targetResourceContainer);
21
22  return true;
23  }
24  }
```

Listing A.1: Example QVT-O method that adds a new assembly context

```
1   helper Set(RuntimeMeasurement) :: checkScalingCondition() : Boolean {
2   self->forEach(measurement) {
3   if (measurement.measuringValue > threshold) {
4   msgAllocation.rootObjects()[MessagingAllocation]->forEach(allocation) {
5   if(allocation.targetResourceEnvironment_Allocation.resourceContainer_ResourceEnvironment->
        asSet()->size() > allocation.allocationContexts_Allocation.
        resourceContainer_AllocationContext->asSet()->size()) {
6   scaleUp(allocation);
7   replicationCount := replicationCount + 1;
8   } else {
9   assert fatal(false) with log('Maximum scaleout factor reached.');
10  };
11  };
12  return true;
13  };
14  };
15  return false;
16  }
```

Listing A.2: Example QVT-O method that checks a scaling condition

A.2. SPECjms2007 measurement results

Message type	Delivery time				
Order	1.67	1.85	1.82	1.7	1.76
OrderConf	1.65	1.71	1.63	1.64	1.69
ShipDep	1.26	1.29	1.24	1.3	1.27
StatInfo	1.26	1.3	1.23	1.3	1.33
ShipInfo	1.35	1.4	1.38	1.35	1.46
ShipConf	0.96	1.11	1.03	0.98	1.1

Table A.1.: Measurement results of SPECjms2007 interaction 1 per message type

Message type	Delivery time				
PriceUpdate	2.46	2.38	2.39	2.32	2.43

Table A.2.: Measurement results of SPECjms2007 interaction 3 per message type