Modelling and Prediction of Event-Based Communication in Component-Based Architectures

Diploma Thesis of

Benjamin Klatt

At the faculty of Computer Science
Institute for Program Structures
and Data Organization (IPD)

Reviewer: Prof. Dr. Ralf Reussner
Second reviewer: Prof. Dr. Walter F. Tichy
Advisor: Dipl. Inform. Christoph Rathfelder
Second advisor: Dr. Ing. Samuel Kounev

Time: 15. May 2010 – 05. November 2010
I declare that I have developed and written the enclosed Diploma Thesis completely by myself, and have not used sources or means without declaration in the text.

Karlsruhe, 2010-11-05
Abstract

With the increasing demand of large-scale systems and the corresponding high load of data or users, event-based communication has gained increasing attention. Originating from embedded systems and graphical user interfaces, the asynchronous type of communication also provides advantages to business applications by decoupling individual components and their processes respectively. However, the possible scalability gained from the event-based communication can result in performance problems in the overall system which are hard to predict by the software architect. Model-based performance prediction is a reasonable approach to predict system characteristics in general. Today’s solutions are however limited in handling the complexity of the additional infrastructure. Especially the impact of many-to-many and asynchronous connections on the overall system is not considered even by advanced projects such as the Palladio Component Model.

This thesis presents an approach to introduce event-based communication in the Palladio Component Model and a transformation to reuse existing prediction techniques. The approach includes an additional automatic integration of an auxiliary repository model. This model encapsulates the characteristics of the underlying middleware infrastructure which is distributed to all event-based connections in the system architecture. An implementation of the approach has been provided as part of this thesis. It was evaluated in a case study based on a traffic information and monitoring system installed in the city of Cambridge. Compared to an existing case study of the same system, the new approach reduced the modelling effort for event-based connections by about 80% and provided more flexibility to test different setups. In addition, the approach reduced the prediction error to less than 5% in most cases.
Zusammenfassung

Durch den steigenden Bedarf an hochskalierenden Softwaresystemen mit großen Anwenderzahlen und Datenmengen, findet auch das Konzept der ereignisbasierten Kommunikation mehr und mehr Beachtung. Während sie früher primär in eingebetteten Systemen und grafischen Benutzeroberflächen eingesetzt wurde, bietet diese asynchrone Art der Kommunikation auch in Geschäftsanwendungen deutliche Vorteile durch die Entkoppelung der Komponenten beziehungsweise ihrer Prozesse.

Allerdings können auch in Systemen mit ereignisbasierter Kommunikation und der damit gewonnenen Skalierbarkeit Leistungsprobleme auftreten, die hier jedoch für einen Softwarearchitekten umso schwerer vorauszusehen sind. Die modellgetriebene Qualitätsvorhersage von Softwaresystemen ist hierfür im Allgemeinen eine sehr geeignete Methode. Allerdings sind die derzeit verfügbaren Lösungen in diesem Bereich eingeschränkt in der Handhabung der Komplexität zusätzlicher Infrastruktur für die ereignisbasierte Kommunikation. Insbesondere der Einfluss der inflationären und asynchronen Kommunikationsmöglichkeiten auf die Gesamtarchitektur wird nur unzulänglich betrachtet.

Die vorliegende Arbeit präsentiert einen neuartigen Ansatz zur Erweiterung des Palladio Komponentenmodells für die explizite Modellierung von ereignisbasiertem Kommunikation sowie die Transformation in ein Modell, das mit existierenden Vorhersageverfahren analysiert werden kann. Ergänzend hierzu wird ein zusätzliches Modell, das die Kommunikationsinfrastruktur beschreibt, automatisch in alle ereignisbasierten Verbindungen der Architektur eingewoben.

Im Rahmen dieser Arbeit wurde eine Implementierung des Ansatzes realisiert und in einer Fallstudie untersucht. Hierbei wurde ein Informations- und Überwachungssystem des öffentlichen Verkehrsnetzes der Stadt Cambridge analysiert. Dabei konnte im Vergleich zu einer früheren Fallstudie gezeigt werden, dass der neue Ansatz den Modellierungsaufwand um bis zu 80% pro ereignisbasiertem Verbindung reduziert und eine höhere Flexibilität für die Untersuchung unterschiedlicher Konfigurationen bietet. Darüber hinaus reduzierte der vorgestellte Ansatz die Vorhersagefehler in den meisten Fällen auf weniger als 5%.
Contents

1. Introduction .......................... 1
   1.1. Motivation .......................... 1
   1.2. Goal of this Thesis .................. 2
   1.3. Structure .......................... 2

2. Foundations .......................... 5
   2.1. Event-Based Systems ................. 5
      2.1.1. Applications .................... 6
      2.1.2. Middleware ...................... 7
      2.1.3. Research Areas ................. 8
         2.1.3.1. Event-Based Communication .. 8
         2.1.3.2. Complex Event Processing .. 8
         2.1.3.3. Distributed Event Processing . 9
         2.1.3.4. Event Modelling Languages . 9
   2.2. Software Performance Engineering .... 9
   2.3. Palladio Component Model ............ 11
      2.3.1. Modelling Roles and Process ... 12
      2.3.2. Meta-Model ..................... 13
         2.3.2.1. Repository Model ............ 13
         2.3.2.2. Composite Model ............ 15
         2.3.2.3. Allocation Model ............ 16
         2.3.2.4. Usage Model ................. 16
      2.3.3. PCM Workbench .................. 17
         2.3.3.1. PCM Ecore Model ............ 18
         2.3.3.2. Graphical Editors .......... 18
         2.3.3.3. Simulation and Prediction Units . 18
         2.3.3.4. Workflow Engine and Completion Framework ... 19
   2.4. Model-Driven Development .......... 19
      2.4.1. Model Transformation .......... 21
      2.4.2. Query View Transformation (QVT) ... 22
         2.4.2.1. QVT-O vs QVT-R ............ 23
         2.4.2.2. QVT-O in Detail ............. 24

3. Related Work ......................... 29
   3.1. Component Models with Event-Based Communication .... 29
      3.1.1. Implementation Oriented Models ... 29
         3.1.1.1. CORBA ..................... 30
         3.1.1.2. SCA ....................... 30
      3.1.2. Analytical Component Models ........ 31
         3.1.2.1. QiImPRESS SAMM ............ 31
         3.1.2.2. CBML ...................... 31
3.2. Performance Prediction of Event-Based Systems . . . . . . . . . . . . . . . 32
  3.2.1. Sachs et al. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 32
  3.2.2. Gorton et al. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 32
  3.2.3. Palladio Component Model . . . . . . . . . . . . . . . . . . . . . . . . 33
3.3. Performance Completion in the PCM . . . . . . . . . . . . . . . . . . . . . 33
  3.3.1. Performance Completion for Message Oriented Middleware . . . . . 33
  3.3.2. Higher Order Transformation (HOT) . . . . . . . . . . . . . . . . . . . 34
  3.3.3. Coupled Model Transformation . . . . . . . . . . . . . . . . . . . . . 35
4. Approach 37
5. PCM Model Extension 39
  5.1. Meta-Model Extension . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 39
    5.1.1. Events . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 39
    5.1.2. Roles . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 40
    5.1.3. AssemblyEventConnector . . . . . . . . . . . . . . . . . . . . . . . 41
    5.1.4. DelegationConnector . . . . . . . . . . . . . . . . . . . . . . . . . . 41
    5.1.5. EmitEventAction . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 43
    5.1.6. Event Handler . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 43
  5.2. Graphical Notation and Editors . . . . . . . . . . . . . . . . . . . . . . . . 44
    5.2.1. Repository Editor . . . . . . . . . . . . . . . . . . . . . . . . . . . . 44
    5.2.2. Composite Editor . . . . . . . . . . . . . . . . . . . . . . . . . . . . 45
    5.2.3. SEFF Editor . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 46
6. Transformation 47
  6.1. Intermediate Event Processing Chain . . . . . . . . . . . . . . . . . . . . . 47
    6.1.1. Event Processing Pipeline . . . . . . . . . . . . . . . . . . . . . . . . 48
    6.1.2. Source Transformation . . . . . . . . . . . . . . . . . . . . . . . . . . 48
      6.1.2.1. Source . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 49
      6.1.2.2. SourcePort . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 49
      6.1.2.3. Distribution Preparation . . . . . . . . . . . . . . . . . . . . . . 50
      6.1.2.4. Event Distribution . . . . . . . . . . . . . . . . . . . . . . . . . 50
      6.1.2.5. Event Sender . . . . . . . . . . . . . . . . . . . . . . . . . . . . 50
    6.1.3. Sink Transformation . . . . . . . . . . . . . . . . . . . . . . . . . . . 51
      6.1.3.1. Event Receiver . . . . . . . . . . . . . . . . . . . . . . . . . . . 52
      6.1.3.2. Sink Port . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 52
      6.1.3.3. Sink . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 52
  6.2. Middleware Platform Specification . . . . . . . . . . . . . . . . . . . . . . 52
    6.2.1. Modelling System Characteristics . . . . . . . . . . . . . . . . . . . . 54
7. Implementation 59
  7.1. QVT-O Transformation . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 59
    7.1.1. Transformation Design . . . . . . . . . . . . . . . . . . . . . . . . . . 59
    7.1.2. QVT-O Experience and Best-Practices . . . . . . . . . . . . . . . . . . 63
      7.1.2.1. Mapping vs. Helper . . . . . . . . . . . . . . . . . . . . . . . . . 63
      7.1.2.2. Main Transformation Dependency . . . . . . . . . . . . . . . . . 64
      7.1.2.3. Code Documentation . . . . . . . . . . . . . . . . . . . . . . . . . 64
      7.1.2.4. Namespace . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 64
      7.1.2.5. Init, Population, End . . . . . . . . . . . . . . . . . . . . . . . . . 65
      7.1.2.6. Limitations of the Eclipse QVT-O Implementation . . . . . . . . 66
  7.2. Integration in the Palladio Workbench . . . . . . . . . . . . . . . . . . . . . 66
    7.2.1. Meta-Model Implementation . . . . . . . . . . . . . . . . . . . . . . . 67
7.2.2. Integration in the Completion Framework

7.2.2.1. Configuration ................................................. 68
7.2.2.2. Completion Execution ..................................... 68
7.2.2.3. Working Copy Job .......................................... 69

8. Evaluation ......................................................... 71
8.1. Goal Question Metric Plan ..................................... 71
8.2. TIME System .................................................... 72
8.3. Evaluation Process ............................................. 75
8.4. Results .......................................................... 75
8.4.1. Q1 Explicitness of the Meta-Model ....................... 75
8.4.2. Q2 Reduced Modelling Effort ............................. 76
8.4.3. Q3 Validity of the Simulation Results .................... 80
8.4.3.1. Scenario 1 .................................................. 81
8.4.3.2. Scenario 2 .................................................. 82
8.4.3.3. Scenario 3 .................................................. 83
8.4.3.4. Scenario 4 .................................................. 84
8.4.4. Q4 Testing Different Middleware Setups/Settings ...... 84
8.4.4.1. Middleware Adaptation .................................... 85
8.4.4.2. Middleware Replacement ................................ 86

9. Conclusion ......................................................... 89
9.1. Summary ......................................................... 89
9.2. Open Issues .................................................... 90
9.2.1. Source Delegation Connector ............................... 90
9.2.2. Complex Event Forwarding .................................. 90
9.2.3. Post Processing Middleware Hook ......................... 90
9.3. Future Development ........................................... 91
9.3.1. Filter Distribution ........................................... 91
9.3.2. Simulation Result Transformation ......................... 91
9.3.3. Probes ........................................................ 91
9.3.4. SOA and SLAs ............................................... 91

A. Case Study Data .................................................. 93
A.1. Scenario 1 ....................................................... 94
A.2. Scenario 2 ....................................................... 95
A.3. Scenario 3 ....................................................... 97
A.4. Scenario 4 ....................................................... 98

Bibliography ......................................................... 101
1. Introduction

1.1. Motivation

Today, software engineering, especially in the field of business applications, is challenged to provide and handle more and more large-scale distributed systems. Independent from the business areas of eCommerce, traffic control, or business intelligence, systems are faced with Internet-scale user groups, interconnected service architectures and large amounts of data that need to be processed with a guaranteed level of service quality. This leads to fundamental changes not only in the technology platforms but also in the architecture paradigms. Even the traditional communication in client-server architectures is challenged by this problem.

The request/response-based interaction is a well known and established approach in this field. It corresponds to the imperative way of programming most developers are used to. While this method stands out because of its simplicity, it comes with scalability limitations caused by the strongly coupled style of interaction [Hoh08]. Concepts like caching or load-balancing have been developed to address this limitation but have not been able to remove all the constraints [MFP06].

An alternative architectural approach which has received increased attention in the last years, is the event-based communication approach. It enables the asynchronous interaction and decoupled execution of system components [Hoh08]. The term “event-based” is taken from the fundamental idea, that one component creates an event that others are able to react to. The crucial point for the asynchronism is the possible decoupling in the dimensions space, time and synchronisation [EFGK03]. These dimensions imply that the event emitting component does not need to know the consumers and any resulting processes. Furthermore, the communication participants do not need to be active at the same time and do not need to align their processing flow with each other. The event-based paradigm provides a completely decoupled and a more scalable concept. The drawback of this paradigm is the higher complexity and a changed data flow given that the data provider becomes the initiator of the communication and not a requesting consumer.

However, as with any approach in software engineering, it is important to build a reliable and well-defined architecture. For event-driven systems it becomes even harder to predict the effects of design decisions. Additional infrastructure such as the communication middleware is introduced with its corresponding impacts. Due to the asynchronism and other characteristics mentioned before, the challenge to estimate the quality properties of
a system gets even more complex. Developing prototypes to gather the required knowledge with a trial and error approach is in general not a feasible solution for the problem. Business, as well as complexity constraints would prevent resources being allocated to this method. As a result, a sustainable concept and a capable tool support are necessary to ensure the reliable development of a sufficient solution.

Even though there is a great demand for such a solution the prediction of quality characteristics of a software architecture is still in its infancy. Some tools are available, especially in the area of performance predictions [Koz09], but they are still not established in the routine software engineering practice. One of the leading software quality prediction tools for component-based architectures is the Palladio Workbench and the corresponding Palladio Component Model [Pal10]. It has a strong focus on both prediction and state-of-the-art modelling techniques. Nevertheless, it still lacks support for asynchronous event-based communication.

Apart from the researches on software architectures in general, there is a large research community dedicated to event-driven architectures with a focus on predicting the characteristics of event stream processing middleware. Internal aspects such as event filtering and distribution are analysed and improved by a number of research groups, among others the Aurora and Medusa projects [SZS03]. But the external view on the impact of event-based communication on the quality characteristics of the overall system is still barely tapped by the prediction community.

1.2. Goal of this Thesis

The goal of this thesis is to enable architects of component-based software systems to model and predict the impact of event-based communication on the performance characteristics of their overall software architecture.

This goal is achieved by extending the Palladio Component Model (PCM) and the corresponding workbench as one of the advanced design-oriented component-based software architecture models in the field of software quality predictions. The existing Palladio meta-model and graphical editors are extended to model asynchronous event-based many-to-many connections between components. Furthermore, model-driven software development (MDSD) techniques are used to transform the new event-based model elements to a quality equivalent model instance based on the existing PCM meta-model. This transformation enables the re-usage of all existing prediction methods that are already included in the PCM workbench. In addition, the transformation contains a completion to weave an auxiliary model into the architecture to specify the characteristics of the communication middleware to be used in the system. The latter also supports the simulation of different middleware products and configurations without changing the system model itself.

The presented approach accounts for the existing component-based software modelling infrastructure of the PCM as well as on-going research in the field of event-based systems [BKR09]. The Palladio Component Model is currently enhanced with the prediction of additional quality attributes such as fault tolerance and maintainability. However, this thesis focuses on the more established prediction of performance characteristics. This also applies to the evaluation of the presented approach.

The advantage gained by an architect from results of this thesis, is the support of his design decisions on event-based communication in a software architecture.

1.3. Structure

The remainder of this thesis is structured as follows: Chapter 2 provides an introduction into event-based systems and performance engineering as well as those parts of model-
1.3. Structure

driven architecture that are required for the presented approach. This is followed by an overview of related approaches in chapter 3.

Chapter 4 discusses the overall concepts of the approach. Chapter 5 describes the meta-model extension in detail as the first part of the contribution. Chapter 6 discusses the model transformation from platform independent to platform specific event-based communication. Chapter 7 describes the details about the implementation of the approach.

The evaluation and its results are then presented in Chapter 8. Chapter 9 gives a conclusion of the thesis and a future perspective.
2. Foundations

This thesis builds on a set of state-of-the-art research fields and technologies. This chapter provides an introduction to the foundations relevant for the presented approach. The contribution of this thesis resides in the field of event-based systems. To better understand their challenges and complexities, the first section will provide an introduction to this area. Although the contribution is an improved performance prediction of these software architectures, an overview of the software performance engineering in general is provided. The Palladio Component Model and the Palladio workbench provide the foundation of the thesis. The important parts of the Palladio project are presented in the third part of this chapter. In the last section, a general overview on the Model-Driven Software Development (MDSD) is provided. MDSD forms the conceptual foundation for the approach of model based performance predictions that is applied in the Palladio project as well as in this thesis. Furthermore, MDSD techniques are used in the implementation of the presented approach.

2.1. Event-Based Systems

Today, the well-established approach of imperative programming is also reflected in the request-reply paradigm that is widely-used in software engineering. This paradigm is based on the concept that one participant, called client, sends a request to another certain participant, called server, and waits for his reply. While he is waiting, he is blocked and not able to perform any other action. This approach leads to strong coupling and dependency. Both are known as sources of problems with quality attributes like performance, maintainability or fault tolerance. This applies for low-level code structures as well as high-level architectures, in particular in today’s trend to service-oriented and distributed systems.

Contrary to this, asynchronous event-based systems interconnect internal components or even multiple systems only by an event that is created by one component and consumed by one or multiple other components. The former, called event source, publishes the event on an intermediary middleware and does not need to care about it any longer. The consuming components, called sinks, are triggered by the events they subscribed for and handle them according to their purpose. An event is described only by its data. It does not include any processing instructions. Therefore, a publishing component neither specifies nor restricts the actions performed by the consuming components when they handle an event.
In the literature, there are numerous of terms used for the elements described above. Sources are also known as event generators, emitters or creators. In this thesis the term source will be used. Sinks are also known as event processors or event handlers. This thesis uses the term sinks for the components receiving the event and the term event handler for the component’s internal processing of incoming events. As the third involved part, the communication middleware is sometimes also known as event bus in the community. While the presented approach is not limited to middleware products that use a central communication bus, the more general term middleware is preferred in this thesis.

The event-based communication approach promises several benefits compared to synchronous request-reply communication [Hoh08]. For example, the asynchronous approach of send-and-forget enables the event source to continue its processing without any need to wait for the connected sinks. This leads to fewer blocking, idle times and resource allocation on the source side. Furthermore, additional sinks can be connected without any changes in the event source. While the only interface definition is the data description of the event send from a source to a sink, the source does not need to check and handle any return values or processing errors of the sinks. This supports a looser coupling and less dependencies between the components as with a traditional request-reply based connection.

Today’s solutions for event-based communication differentiate each other in the decoupling of the components. Eugster at al [EFGK03] defined the decoupling dimensions time, space and synchronisation. In this definition, time means that event producers and event consumers do not need to be active at the same time. Space means that they do not need to know each other. For example, a producer does not know who is going to consume his events and a consumer might not worry about who has produced an event he is interested in. Synchronisation as the last dimension means that the producer does not wait for the consumers to process an event. If producers and consumers are decoupled in this dimension, the producer can continue his own process directly after emitting a new event.

However, event-based communication is associated with drawbacks that also need to be considered. It introduces additional complexity to a software system [Hoh08]. The additional middleware for the event transmission implies an additional source of failure and requires additional resource demands for its internal processing. Furthermore, predictions of the components influence on each other becomes even harder in this loose relationship. A factor that should not be disregarded is the smaller group of engineers used to design and work with event-based systems compared to those that are used with the traditional request-replay paradigm [MFP06].

2.1.1. Applications

Event-based communication is not related to a specific type of applications. Due to the general concept, it could be applied to embedded, desktop and even client-server applications.

Up-to-now, embedded systems might be one of the most typical applications for event-based communications. Sensors like traffic controls, temperature probes or gas analysers send their measurements into the system and the software components process them. For example, they can react on critical values and generate an alert or log the information. Actually, in some domains of embedded systems, there are already specifications including event-based communication like PROFIBUS ([EP02]) as an international standard for field buses.

A typical field in desktop applications are Graphical User Interfaces (GUI). Any user interaction such as mouse clicks or keyboard inputs produce events published by the according
GUI components. Event handlers that have registered on specific GUI components are then triggered to perform an implemented action. The most programming platforms provide libraries for this kind of GUI development such as Windows Forms in the Microsoft .Net framework or Swing in the Java platform.

A still young domain of event-based communication is the field of distributed business applications and Service Oriented Architectures (SOA). A typical scenario are tickers for financial information in the stock market. Recently event-based communication is used more often in internet-scale applications such as document processing in search engines or information publishing in logistic and retail systems.

2.1.2. Middleware

Nowadays, the development of a software with event-based communication, does not require to build all the required infrastructure from scratch. A full range of out-of-the-box middleware products is available on the market to cater for this need. These middleware products offer interfaces to hand-over events from the source to the middleware and later from the middleware to the target sinks.

There are two general groups of these communication middlewares. One group includes systems with a peer-to-peer architecture and the other contains systems with a centrally deployed middleware. In the former group, the middleware provides components installed individually for every communication participant with direct connections between each other. In the latter group, they use a centrally installed communication hub as known from Java Messaging Service (JMS) solutions such as ActiveMQ, WebSphereMQ, or JBoss messaging. Nevertheless, all of them provide various features and characteristics such as routing strategies, support for topics and filters, transmission protocols, guaranteed delivery, and numerous others.

Depending on the system requirements and target environment, a software architect can choose from a range of middleware products that implement different concepts and characteristics.

Message Oriented Middleware

Message Oriented Middleware (MOM) is the general term for software infrastructure that delivers messages between multiple systems or subsystems. These infrastructures provide a message queue and an interface to hand-over messages from the publisher to the middleware and later from the middleware to the appropriate recipients.

In practice the terms message and events are often used interchangeably. Messages are a communication medium and events are a communication style. Strictly speaking, messages can be used to deliver events and are therefore prone to be mixed up.

The most significant difference between messages and events is the fact that messages are a communication infrastructure technology and event-based communication is a communication paradigm. The latter one might be associated with a message oriented middleware which asynchronously transmits the event as a message to a set of subscribers who are not necessarily known by the publisher.

Event-based communication is often referred to as a publish-subscribe concept in the field of message oriented middleware.

There are numerous established open-source and commercial Message Oriented Middleware solutions. For example, the JAVA platform provides a specification named Java Messaging Service (JMS) \(^{[HBS+02]}\) with different implementations available such as ActiveMQ or JBoss Messaging.
Peer-To-Peer and Central Architectures

A significant difference between middleware products is the architecture of the communication components and their distribution.

On the one hand, there are systems with a central architecture using one or more explicit middleware components that can be deployed on distinguished nodes of the infrastructure. Most of the message oriented middleware systems described before use this concept.

On the other hand, there are peer-to-peer systems without a central messaging component. Each participant in the communication has its own middleware wrapper that is able to communicate directly with the middleware wrapper of other components. Generally these wrappers are deployed on the same infrastructure as the participants themselves. This approach is often used in embedded and widely distributed scenarios as in the traffic monitoring system of the TIME project [BBE+08] used in the evaluation of this thesis.

Both of these concepts have advantages and disadvantages. The additional infrastructure and possible single point of failure in the central approach might be feasible in one case while the peer-to-peer approach with a more distributed source of failure might be better in another one, even if it requires a more complex coordination for more than two participants.

2.1.3. Research Areas

A comprehensive term often used by the researches of event-based systems is the “Event-Driven Architecture”. It is generally used if the event paradigm is implemented in a software system. The fundamental characteristic of the event-driven architecture is the style of communication as was already mentioned before. There are several subtopics in the research on event-driven architectures to discover the resultant challenges. These include the complex event processing and the distributed event processing as well as the event-based communication which is the focus of this thesis. The following subsections provide a brief introduction to the most relevant subtopics.

2.1.3.1. Event-Based Communication

Event-based communication is a type of communication protocol. The crucial aspect of event-based communication is the unidirectional and asynchronous communication between the participants and their loose coupling. The impact on the changed system design in relation to complexity and extra-functional properties forms the focus of this part of the research on event-driven architectures.

2.1.3.2. Complex Event Processing

Complex event processing is a widespread research topic in event-driven architectures. It concentrates on the design of the events themselves and the related processing effects [Luc01]. This topic is based on the questions how much information should be included in an event, how should it be structured and how could it be processed efficiently. The event content can range from a single bit up to complex data types or hierarchical, convoluted events.

This thesis focuses on the impact of event-based communication on the overall system architecture. In this context, the complex event processing is treated as a characteristic of the data and the middleware represented in the model. An architect who is working on a system that processes complex events can specify the quality relevant aspects as part of the model. He is able to specify the data structure of the events and the related resource demands as they have an impact on the quality properties. The corresponding resource demands of the middleware components and the event handlers can also be specified in the appropriate models.
## 2.1.3.3. Distributed Event Processing

Distributed event processing is another major topic in the research area of event-driven architectures. It is focused on systems with a complex communication infrastructure between the interacting participants [MFP06]. It particularly targets middleware with a set of communication nodes involved.

As with the complex event processing, the influence of the distributed event processing is treated as an aspect of the middleware in this thesis. The performance relevant characteristics can be specified as part of the middleware model and are therefore not directly handled in this thesis.

### 2.1.3.4. Event Modelling Languages

In the field of event-driven architectures a couple of domain specific languages have been published. Most of them are low-level processing languages such as query languages comparable to the Structured Query Language (SQL) [Ins08] for relational databases. One of them is the query language developed in the Rapide project at the Stanford University [RDTVC97].

There is only a small number of languages specific to software architectures that provide meta-model elements for event-based communication. One of them is the Unified Model Language (UML) [Obj10b] which provides Signals as model elements with characteristics comparable to events. But even these signals are insufficient to model all aspects of event-based communication. For example, the signals are specified as part of a behavioural description. They do not provide any facilities to model an event-based connection in a component model.

## 2.2. Software Performance Engineering

The extra-functional properties of software products such as their performance and reliability are known as crucial factors for their success since a long time. A number of serious software problems caused by insufficient software performance exist and have been officially published [Gla98]. But even in everyday work, less serious problems repeat themselves. In the past and at present, these problems are often targeted with a trial and error approach but this can be insufficient and become too expensive in matters of business constraints and scale of modern business applications [WS03]. As already established in other engineering disciplines such as engine or bridge construction, it is the goal of software performance engineering to assure the extra-functional properties of software systems even before they are implemented.

In the mean time, performance engineering is one of the long-standing research topics in information science. In the beginning researchers worked on analytical models such as Queuing Networks [BGdMT98], Stochastic Petri-Nets [BK02] and Stochastic Process Algebras (SPA) [HHK02]. Common to many of these approaches is the fundamental concept of the semantic of generalized semi Markov chains [MB8a]. Analytical as well as simulation based predictions of these models are available and can be chosen depending on the concrete assumptions and required analysis. These type of models are not limited to software systems but target a general application of performance engineering.

All of the approaches presented above require a fundamental knowledge of the theoretical concepts, are limited in their usability for complex systems or make unrealistic assumptions on the analysed system. Furthermore, most of the software architects and developers are trained to use design-oriented languages such as the Unified Modelling Language or at least parts of it. Only a few of them might be familiar with the theoretical and analytical models and their applications [Bec08a].
Smith et al. first established the term *Software Performance Engineering* (SPE) and introduced an additional focus on software design to the performance engineering research [Smij90]. With the evolution of the software development itself and the change from low-level software design to architectural level approaches, even the software performance engineering evolved to more design-oriented analysis. This includes additional profiles for the UML such as the UML profile for Scheduleability, Performance and Time (UML-SPT) [Obj05b], the UML profile for QoS [Obj05a], and the UML-MARTE profile [Obj06c].

In addition, the OMG defined a general process for model-based performance predictions as presented in Figure 2.1. The starting point of this process is a model that describes the software system itself. This model can be created with established modelling languages such as UML. The modelled software might already exist but does not need to. The general software model does not include any information specific to the performance characteristics of a software. This information is added in the next step. If the software does not exist yet the resource demands are estimated. If the system is already available, for example in the case of a refactoring, measurements of the existing software can be used to gather the relevant information to annotate the model. The annotation can be done using one of the UML profiles mentioned before or using a meta-model designed specifically for this purpose such as the KLAPER model [GMS05]. Following on, the annotated software model is used as an input for a transformation into an analytical model. The analytical model can then be analysed or simulated by standard tools in this field. As a final step, the prediction results are returned as feedback for the original software model.

![Figure 2.1.: OMG Performance Prediction Process](image)

Due to the still emerging field of model-based performance prediction there is no standard model or standard tool for the specified process. Most of the available models and tools differentiate from each other in the performance factors examined by their model annotations. The performance specific meta-models also vary in their alignment with the UML. For example, the domain specific language of the SPEED tool developed by Smith et al. [Smij90] has no overlap with the UML, while the Palladio Component Model in contrast is aligned with most common UML models. The third aspect in the differentiation is the available tool support which is only available on an individual basis for the current models.

The following list provides an overview of performance prediction approaches in the context of component-based architectures and is related to the approach of this thesis. The presented approaches have been selected because they are all employable before the software is actually implemented. A complete overview of performance models even behind
the scope of this thesis can be found in the surveys of Becker et al. \cite{BGMO06} and Koziolek et al. \cite{Koz09}.

- **RESOLVE-P** \cite{SKK01} uses the Big-O notation as foundation of their approach. It is used to describe the time and memory consumptions of components in relation to the input parameters. Individual Big-O notations are used to describe the single components. A composition of these Big-O notations is then used for a composite systems build on top of them.

- **PACC** \cite{HMSW02} developed as a conceptual framework to combine the performance properties of certified components and their assemblies. The weak point in this approach is that it provides only a conceptual framework and the quality of the prediction relies on the concrete method used for the prediction of the properties.

- **CB-SPE** \cite{BM04} applies the original SPE method of Smith et al. to component-based systems but makes the restriction to ignore the impacts of the internal processing and input parameters. Only probabilistic resource demands are used and the dependency on input parameters is ignored completely.

- **CBML** \cite{WW04} presents an approach to use Layered Queuing Networks (LQN) to describe components from a performance properties point of view. Assembling a set of components is reflected in the assembly of the appropriate LQNs. The combined LQN is then solved to achieve the overall performance properties of the system. In addition, they introduced the concept of completions to bring in performance details of specific subsystems by automatically adding more detailed LQNs for them. With this process, Wu and Woodside achieved a more precise performance prediction \cite{WPS02} and established the term “completion” for such proceedings.

- **ROBOCOP** \cite{BdWCM05} is focused on the area of embedded systems. It is used to describe component internals in relation to the parameters of external services and resources. The drawback of ROBOCOP is its limitation in the description of the environment. Due to the focus on embedded systems there are constraints in that resource parameters can only be specified as constant values and software layers are not supported at all.

### 2.3. Palladio Component Model

The development of the Palladio Component Model (PCM) started in 2003 at the University of Oldenburg, and since 2006 it has been further developed at the Karlsruhe Institute of Technology (KIT) and the FZI Research Centre for Information Technology. The PCM is accompanied by a workbench that is based on the Eclipse platform. The workbench provides graphical editors aligned with the UML syntax, a simulation engine as well as several transformations into analytical prediction models and different visualizations of the prediction results.

The Palladio Component Model is one of the most evolved solutions in the field of model-based performance prediction of software architectures. As it includes modelling as well as prediction facilities, it perfectly serves as a foundation for the presented approach and removes the necessity to develop a completely new solution from scratch.

In the following a short overview of the most significant parts of the PCM is provided. Further details are given in \cite{BKR09} and can be found on the projects website \cite{Pal10}.

As indicated by the name, the Palladio Component Model has a focus on component-based architectures. From the PCM point-of-view, a systems quality and especially its performance characteristics are influenced by four factors as presented in Figure 2.2. The
implementation is the most obvious criteria, as it describes the design of the actual software. If the implementation makes use of external components, those required components can also have an impact. Both the software itself and the required components will need to be deployed on an infrastructure. The required resources provided by the infrastructure are the third important criteria with an impact on the overall system quality. Finally the system usage has an important impact. For example, a system may provide a sufficient performance for a small user group even with a huge number of request, but it may be totally overloaded by a large user group even with a small number of requests per user.

Figure 2.2.: Performance Impacts

The clear separation between the different architectural aspects as described above is a common technique widespread in the software architecture and not only limited to component-based systems. It can be found in general disquisitions on quality in software architectures such as the separation between application, technical and IT architecture as described in [Sie06].

2.3.1. Modelling Roles and Process

The Palladio Component Model specifies roles and a corresponding process for the development of a component-based software architecture. There are four roles involved in the Palladio approach:

- **A Component Developer** is responsible for describing a component in a grey box approach. This means, that he does not describe the implementation in detail as in a white box approach. He has to nevertheless provide a description of the performance relevant aspects of the component internals. The system prediction takes this information into account to achieve more precise results than a clean black box approach would allow.

- **A Software Architect** assembles components to define a system to satisfy the functional and extra-functional requirements. If one of the required components is not available upfront some interaction with the component developer is necessary. Depending on the applied development strategy, either the architect provides a brief description of the missing component to be created by a Component Developer or the architect also acts as a component developer and creates the complete component by himself. In the first case, the architect should describe at least the interfaces to be provided by the new component. In the PCM, this is explicitly called a providing component type. A component developer then provides a so called complete component type that fulfils the architects requirements.
A **System Deployer** is involved in the PCM process when the completely described system is to be deployed on an infrastructure. He provides the infrastructure description with the required resources. Furthermore, he allocates the components to the available infrastructure nodes.

A **Domain Expert** provides a reasonable description of the estimated system usage. This includes the information on how many users will use the system and what their behaviour might look like.

How many people are acting in these roles is not defined by the PCM but the separation enables a better understanding of the different tasks and their individual complexities. This enabled the development and provision of domain specific languages (DSL) for each of them as part of the PCM meta-model.

### 2.3.2. Meta-Model

The PCM provides a meta-model to describe component-based architectures of software systems including their performance related criteria as described above. As illustrated in Figure 2.3, an instance of the PCM meta-model is described with domain specific languages appropriate to the four roles Software Architect, Component Developer, System Deployer and Domain Expert.

![Palladio Meta-Model and DSLs](image)

#### 2.3.2.1. Repository Model

The major entities in a PCM repository are the **Component** and **Interface** elements. Both are first class entities and can exist on their own. The links between them specify the **RequiredRoles** and **ProvidedRoles** that identify interfaces as required or provided interfaces of a component. While the former describe the services a component might need to fulfil its purpose, the latter describes the services a component provides by itself. The interfaces of a component form the contract how the component can be used.

In the PCM meta-model, the provided and required roles are linked to a component by a reference from the InterfaceProvidingRequiringEntity to the specific role element. Repository components are subclasses of this entity type as shown in Figure 2.4. The separation between providing and requiring entities is related to the distinction of providing, requiring and complete component types as part of the PCM development process.

Interfaces are used as a contract for the services provided and required by a component. As shown in Figure 2.5, interfaces consist of a set of signatures that describe the operations provided or required by a component in detail. A signature references to a set of parameters to represent the input parameters of the specified operation. Parameters again are
described by their data type. In addition to the parameters, a signature has an optional reference to a data type element that describes the return value of the signature.

Interfaces on their own are sufficient for a black box view of the components. In addition, the PCM’s grey box approach requires a description of the components internal processing from a quality prediction point of view.

To satisfy this requirement, the component developer describes the internals with so called Resource Demand Service Effect Specifications (RDSEFF). RDSEFFs are an extension of Service Effect Specifications (SEFF), which themselves define an abstraction of the components internal behaviour. RDSEFFs add information about the quality of service relevant aspects. Those aspects include direct resource demands, calls to external services.
and the data-flow aspects with the necessary level of detail.

The PCM describes RDSEFFs aligned to the activity model of the Unified Modelling Language. It provides a sequence of actions with individual predecessors and successors to model the transitions from one action to another. The different type of actions can be grouped into data-flow actions and processing actions.

The later includes an InternalAction type to describe resource demands as an abstraction of the internal processing. It describes a resource demand as a stochastic expression that can include references to variables within the context of the action. Beside the InternalAction, a SetVariableAction enables the component developer to set the characteristics of a variable as part of the actual processing.

The data-flow actions include the ExternalCallAction which is able to set variables as input parameters for the called service. Beside this, BranchActions and ForkActions are also available. Both of them describe zero, one or more sub behaviours as included RDSEFFs. The BranchAction can use guarded and probabilistic transitions to determine the sub-workflow to be executed. The ForkAction executes all sub-workflows and does not need to synchronize with them. This means, that the processing is not blocked until all sub-workflows are finished.

There are a couple of more actions available in the PCM but these are less relevant to this thesis.

When variables are specified or processed in the PCM it is done with a focus on their characteristics. For example, if a variable is about a message object, the exact value of the message might not be of interest but the byte size of its content is relevant for the system performance. When a component developer describes these characteristics he creates a VariableUsage which references the variable he would like to describe and includes VariableCharacterisation elements to define all required characterisations of the variable. The appropriate actions provide a reference to a set of these VariableUsages.

2.3.2.2. Composite Model

The components and interfaces of a repository provide the building blocks to model a concrete system. To do so, the PCM provides a composite model to describe complete systems as well as composite components. All of them are so called composed structures in the PCM and can be specified as an instance of the composite model.

As shown in Figure 2.6 a composed structure contains AssemblyContext elements. Each of these AllocationContexts represent an instance of a repository component. The optional VariableUsage that can be specified for an AssemblyContext is used to set instance parameters of the component. The composite model provides AssemblyConnectors to assemble multiple component instances. Each of these reference to two AssemblyContexts and represent the link between the RequiredRole of the first instance and the matching ProvidedRole of the second instance.

All composed structures, regardless of their specific type, need at least one provided interface to make any sense at all. They are therefore subclasses of the InterfaceProviding-RequringEntity and able to reference provided and required roles. Composed Structures are slightly different to repository components in how their internal handling of a provided role is specified. They do not contain RDSEFFs to describe the processing of a provided role. Instead DelegationConnectors can be used to link between internal and external roles. A ProvidedDelegationConnector is available to specify an internal component that should handle the processing of the external provided role. In the same way, a RequiredDelegationConnector links an internal required role to a compatible external required role.
2.3.2.3. Allocation Model

The PCM provides an allocation model for the system deployer. This model is used to allocate component instances of a system to the infrastructure nodes of a resource model. To do this, the system deployer places an allocation context in an infrastructure node and specifies the component instance to be allocated by this context.

The PCM also supports distributed systems in the allocation model. In such a case, component instances of a single system can be placed on different infrastructure nodes. The nodes themselves can be connected with linking elements that represent the network resources between them [RBK+07].

2.3.2.4. Usage Model

The PCM provides an usage model to support the role of the domain expert. An instance of a usage model is able to describe multiple usage scenarios, comparable to the UML use cases. Figure 2.7 shows that a usage scenario contains a workload definition which can either be an open or a closed workload. In this context “open” means that the workload specifies the rate of incoming requests such as “every 5 seconds a new customer arrives using the system according to the described scenario”. This however does not make any statement about the maximum number of concurrent users in the system. In the worst case scenario this could raise indefinitely. In contrast a “closed” workload is applicable for systems with a closed user group where the maximum number of users can be declared.

The closed workload is specified as the total amount of users in the system and their individual think-time between a reply to one request and the point in time when the user sends the next one. For example, a supermarket scenario with a fixed number of cashiers with a limited performance to enter products into their cash desk is a closed workload. In the PCM users described by their workload are not limited to human operators. They can also represent any type of client system.

Besides the type of workload, a usage scenario describes the user’s interaction with the system as a ScenarioBehavior. Comparable to the definition of a components internal processing with an RDSEFF, the description of how users interact with the system is also aligned with the UML activity model. As illustrated in Figure 2.8 this scenario behaviour can represent an interaction-flow including loops and branches. Every direct interaction
with the system is represented by an EntryLevelSystemCall. Such a call can include a
VariableUsage to describe the expected user input for the system call as already known
from the repository model elements.

2.3.3. PCM Workbench

The Palladio project has developed a workbench to support the work with the Palladio
Component Model and to perform predictions for PCM model instances. The workbench
is a desktop application based on the Eclipse platform that includes graphical editors for
the provided domain specific languages. Furthermore, it provides support for a number
of different performance analysis techniques including layered queuing networks [KR08],
stochastic process algebras [Hap08] and simulation models [Bec08a]. All of them accept a
PCM model instance as input and simulate or analyse this model.
2.3.3.1. PCM Ecore Model

The PCM workbench extensively uses the infrastructure of the Eclipse platform. This also applies to the implementation of the PCM meta-model. The eclipse modelling project provides the Ecore infrastructure as an implementation of the essential meta object facility specified by the OMG [Obj06a]. The meta-model is therefore implemented on a well defined base with advanced tool support for editing, code generation and transformation.

2.3.3.2. Graphical Editors

One of the intentions of the Palladio project is to push software development as an engineering discipline. This includes an architecture level modelling language for software quality predictions but also needs to consider the usability of the provided workbench. These requirements led to the development of a graphical notation for the presented meta-model. In order to reach a better acceptability and a smoother learning curve, the graphical notation was developed in alignment to the UML2 notation. Established notations for component, class and activity diagrams are reused in the repository, composite, usage model and SEFF editors.

The graphical editors also make use of the eclipse modelling project and are partly generated using the Graphical Modelling Framework (GMF). With a focus on usability, the editors hide some details of the meta-model and include helpers to automatically create elements to support the software architects work.

2.3.3.3. Simulation and Prediction Units

Research focusing on software quality analysis, simulation and prediction already developed a wide range of approaches. The PCM workbench includes four different approaches from analytical solving to prototype code generation.

- The **Analytical Solver** provides the fastest prediction feedback based on mathematical calculations of the prediction results. As a drawback, the solver is limited to a single user scenario and has limitations in the complexity of the analysable model.

- The **SimuCom Simulation** generates code for a simulation library equipped with time consumptions and probes for the measurement. It is then executed by the workbench. The results are stored and the simulation code is removed automatically afterwards. The simulation provides more realistic results than the solver. It is able to simulate a multi-user scenario and at least in theory, it is able to cover the full complexity of a system. As the simulation executes and to measures real code, it takes longer than the analytical solver to provide the prediction results. Furthermore, modelling the whole environment and all performance relevant factors would require great effort. The analyst has to find a good trade-off between the considered details.

- The **Prototype Based Prediction** generates the code for a prototype of the modelled system. This prototype can be deployed and executed in a real environment. This approach releases the software architect from modelling all the environmental specifics. As a drawback it requires him to set-up the environment to run the prototype. Nevertheless, this prototype is still an abstraction of the real software and the results are still an approximation.

- The **Generation of a Software Skeleton** provides the starting point for the developers to write the real software. Of course, executing the final product would indicate the real quality of service. But obviously this approach requires a full software development and is expensive as well as time consuming.
At the time of writing, the PCM workbench provides only these four types of performance predictions with different capabilities as described above [BKR09]. There is ongoing research on transforming PCM instances into another meta-model that can be predicted based on Queuing Petri Networks. This approach might be able to provide faster prediction results but only for restricted PCM instances.

2.3.3.4. Workflow Engine and Completion Framework

Thanks to the well-defined PCM meta-model and the Ecore environment, the PCM workbench makes use of model-driven techniques to predict the quality attributes of PCM instances. The prediction process is implemented using a workflow engine integrated in the PCM workbench. Figure 2.9 illustrates the general workflow of the SimuCom simulation. As a first step, the PCM models are loaded into a blackboard provided by the workflow engine. This blackboard is accessible by all jobs to read from and store models on it. The PCM models are validated in the second step. This includes the validation of OCL constraints defined in the meta-model for example. Afterwards any configured completions are executed. The models are now prepared and the remaining steps perform the simulation itself. In the fourth step a temporary working project is created in the eclipse workspace. The following job generates the simulation code for the PCM model and stores it in the working project. Using this simulation code, the next job starts the simulation engine. The results are stored in a previously configured data store and can be visualized and evaluated at a later stage. Optionally, the results of the working project can be cleaned up by the workbench or sustained to check up the details of the simulation code.

The completion step performed within this workflow makes use of a completion framework provided as part of the Palladio project. This approach executes transformations on a PCM model to add additional information before the hand-over to the simulation. The framework is integrated in the configuration of the SimuCom workflow. Before the workflow is started, the analysing architect can choose one or more completions to be applied to the PCM model. The PCM provides some pre-installed completions he can choose from or he can configure his own ones. The completions make use of techniques of the model-driven development.

2.4. Model-Driven Development

With the growing size and complexity of software systems, modelling languages have been introduced to support a better documentation and communication of software design. One of the most well-known representatives is the Unified Modelling Language (UML). It has become a de facto standard in the software industry. Using software models enables the architects and developers to focus on the software structure and to fade out low-level implementation details.

Model-driven development is an approach to handle the grown complexity in the software development by focusing on models instead of programming. Models are therefore no
longer documentation instruments but first class entities in the development process and sometimes during application runtime as well. The Object Management Group (OMG) has developed a methodology named Model-Driven Architecture (MDA) [Obj03] and a set of specifications around this topic. In this methodology, software development makes use of different types of models and differentiates between platform independent models (PIM) and platform specific models (PSM). While the first one focuses on the domain processes and structures, the later takes implementation and platform specific information into account.

A crucial point in the MDA approach is not just to talk about source code generation based on software models but to also transform between different types of models too. The later can be done between PIM and PSM, but also between PSM and PSM or PIM and PIM. The transformation from PSM to PIM is rarely used, but can also be used to support software reverse engineering or analysis. The commonality of all these alternatives is the necessity of a well-defined meta-model to rely on. The OMG has specified the Meta Object Facility (MOF) [Obj06a] for this purpose and in version 2.0 the specification describes the separation of meta-models and models. A meta-model can be considered as a formal language to describe a model. Nevertheless, a model itself can be a meta-model for another model. In theory this chain could be infinitive. Figure 2.10 illustrates an exemplary stack of four model layers, M0 to M3, to describe a software. M0 represents a running instance of the software. M1 is the class model of the software. The layer M2 contains the Unified Modelling Language (UML) as a meta-model. It specifies the language to build the class models on M1. The upper-most layer M3 is the MOF meta-model itself. It provides basic modelling concepts such as Classes, Attributes or Associations to model meta-model languages such as the UML. It is sometimes referred to as a meta-meta-model and the meta-model of the MOF language is the MOF meta-model itself.

Figure 2.10.: MOF Layer Example
2.4. Model-Driven Development

The model-driven development already covers areas exceeding the technical software implementation. While it abstracts as much as the architect would like to, it can also be used in discussions with domain experts to design business processes or other applications. This is also applied to the Palladio Component Model. For example, the domain expert who specifies the usage model does not need to know anything about the software itself except of the external provided system services.

2.4.1. Model Transformation

As described before, the model-driven development differentiates between models and meta-models and a well-defined meta-model is necessary to develop model transformations. These transformations can be applied not only to a specific model instance but to all instances described with the same underlying meta-model.

Czarnecki and Eisenecker published one of the fundamental papers on model transformation characteristics [CE00]. They analysed this topic in detail and provided a feature tree summarizing the possible shapes.

In general, there are two major types of model transformations. One type is about the transformation from one model to another, the other type transforms models to textual representations.

Model-to-Model Transformation

This type of transformation takes one or more model instances as an input and generates one or more models as an output. It is also possible to only modify the input models, which is referred to as in-place transformation. There are no restrictions on the meta-models of the in- and output models. They can either be different or all of the same type.

Looking at a meta-model as a language, the model-to-model transformation is comparable to natural languages in general. For example, if someone translates an English text into German, it is comparable to a transformation from one meta-model to another. In the model-driven software development, a common example is the transformation of a UML class model to an ER model for relational databases.

In contrast to this, if someone rewrites an English poem to an English novella it is comparable to a model-to-model transformation within the same meta-model. In the model-driven development this is often illustrated with the transformation of a platform independent model to a platform specific one and introducing additional information.

Model-to-Text Transformation

Model-to-text (M2T) transformations use a generator to produce textual artefacts based on a model. Furthermore, it makes use of a well-defined meta-model to describe the output which should be generated for specific meta-model elements. To keep the analogy of the everyday languages, it is comparable to someone writing down a novella as a book. In the model-driven development, the standard application of model-to-text transformations is the generation of source code and documentation. Beside of this, every textual artefact can be generated by a model-to-text transformation and used even to write a model to a storage.

The generation of software with M2T transformation techniques is a research topic on its own. The original request to generate complete software from a model has turned out to be only possible for a limited number of systems. Most software systems contain parts with algorithms or customisations that require manual written code or the modelling might be less efficient than the manual coding itself. To overcome this obstacle, the model-driven
development approach distinguishes between platform, individual and schematic repetitive code. As illustrated in Figure 2.11, only the repetitive code is a reasonable target for code generation based on M2T transformation.

The model-to-text transformation is also applied in the PCM workbench to generate the simulation source code. In this instance the open source framework openArchitectureWare is integrated in the workflow. This framework provides an advanced transformation language named Xpand for model-to-text transformations [BGG+07].

2.4.2. Query View Transformation (QVT)

The OMG has developed a set of specifications in the field of model-driven architecture and development. To specify transformations between two models, the OMG published the Query View Transformation (QVT) language [Obj10a].

- **The Query** describes the request to the source model to identify the part of the model that should be changed.
- **The View** describes how the results of the query should be presented in the target model.
- **The Transformation** is the process to get from the Query to the View.

QVT supports a relational as well as an operational description of transformations. The former is a declarative approach and describes the expected results of the transformation and the later is a more imperative style and defines how to achieve the results.

As shown in Figure 2.12, QVT specifies a language core. The declarative language of QVT is named QVT Relational (QVT-R) and is built on top of this core. The operational language of the specification is called QVT Operational (QVT-O) and is built on both, the QVT-R language and the core. If these languages are not powerful enough, one is able to develop custom black box language extensions that use either QVT or other programming languages such as JAVA or C++.

To navigate the model and to describe any model constraints, QVT does not invent a new approach but uses the Object Constraint Language (OCL) specified by the OMG [Obj06b]. This language is aligned to state-of-the-art programming languages to access model elements and their attributes. It is also able to select sets of elements with conditional statements.
2.4.2.1. QVT-O vs QVT-R

Both language derivations are still young and the number of reference projects and the available experience reports are still rare. To choose one or the other has a big impact on the development of a transformation. It is necessary to consider the advantages and disadvantages of both alternatives.

The experience of existing projects, in the context of the PCM and a study presented at the Sixth International Conference on Information Technology [GPT09], have been summarized in a list of characteristics of the two languages in comparison to each other. The characteristics are presented in the following.

Transformation Organization

Complex transformations can grow to a reasonable amount of code. As with other programming languages, it often makes sense to split the transformation into multiple files for better maintainability and re-usability. Only QVT-O supports the organization of a transformation into multiple files. QVT-R requires having all transformations within a single file.

In-Place Transformation

Using a model-to-model transformation to change an existing model and to not generate a new instance can be useful in workflows especially if a pipeline pattern is used as in the PCM completion framework. QVT-R does not support in-place transformations by nature and would require a transformation to create a complete copy of the model [KG09]. QVT-O directly supports in-place transformation as well as copy operations for model elements and sub-models.

Branching

QVT-R only supports simple branching. This means, that the if-then-else-endif only allows one “if” and one “else” expression. This would result in multiple nested if-then-else constructs if more than a binary branch decision is needed. Instead QVT-O supports an infinitive number of branches with an else-if construct (elif). If possible, a QVT-R based transformation should be designed in a way that prevents the necessity of “else if” constructs.

Transformation Control

Non-trivial transformations require a fine control of the transformation as it is provided by QVT-O. For example, QVT-O is able to iterate over a list of elements and invoke different actions with each iteration. In such a case, QVT-R requires to write multiple relations and combine them into one consistent sequence, which is much more complex than the QVT-O concept.
Required Programming Skills

Nowadays most programmers are familiar with imperative programming languages but only a few are used with declarative approaches. The chance to find a programmer who is able to maintain a QVT-O transformation is therefore higher than for QVT-R.

Graphical Representation

QVT-R provides only one possible way to write relations \[\text{[GPT09]}\] and therefore it has been possible for the OMG to develop a graphical notation as part of the UML. Such a graphical notation is still missing for QVT-O. Nevertheless, QVT-O is comparable to other imperative languages and therefore existing notations for these languages can be reused.

2.4.2.2. QVT-O in Detail

This thesis makes use of the QVT-O language due to the advantages it provides for the presented approach \[\text{2.4.2.1}\]. Furthermore, it provides transformation specific facilities, such as model navigation and element selection which provide more power compared to common programming languages like Java or C#.

Main Transformation Declaration

The main element of a QVT-O transformation is obviously called “transformation”, and as shown in Listing \[\text{2.1}\] it is identified by the protected keyword \textit{transformation} and followed by the name of the transformation.

\begin{verbatim}
Listing 2.1: QVT-O Transformation Definition
import myUtilityLibrary;

modeltype UML uses SimpleUml
  ( "http://omg.qvt-examples.SimpleUml" );
modeltype RDBM uses SimpleRdbms
  ( "http://omg.qvt-examples.SimpleRdbms" );

transformation exampleTransformation(in inputModel : UML,
  out outputModel : RDBM)
  access library myUtilityLibrary;

main() {
  ...
}
\end{verbatim}

The parentheses contain a comma separated list of models that are processed in the transformation. Each model in the list is prefixed with an identifier for its accessibility. The possible identifiers are “in” for “input only”, “out” for “output only” and “inout” for “input and output” at the same time. An exclusive input model provides only read access for the transformation. An output model is only used as target model and the inout model identifies a model that can be used in an in-place transformation with transformations applied directly to the input model. The models can be placed in this list without any need to order them in relation to the access modifiers. Obviously the models are identified with an unique name to access them within the transformation. As a third part, the meta-model of each model is specified as a prefix separated by a colon.

The available meta-models in the QVT-O transformation need to be declared at the top of the transformation script. This declaration can be a reference to a meta-model as shown in listing \[\text{2.1}\]. The reference is identified by the keyword “modeltype” and followed by the
identifier of the meta-model to reference it in the transformation. There are two options to specify the meta-model. The first uses an identifier (i.e. SimpleUML) and a reference to the meta-model if it is not already available in the transformation environment. The alternative option specifies a meta-model as an in-line declaration as sketched in listing 2.2. The latter renders the meta-model declaration unusable in other transformation scripts. Referencing an external meta-model is therefore the preferable option.

Listing 2.2: In-Line Meta-Model Declaration

```plaintext
metamodel SimpleUML {
    abstract class UMLModelElement {
        kind : String;
        name : String;
    }...
}
```

External QVT-O Libraries and Transformations

Listing 2.1 includes an import declaration and a transformation signature with the definition `access library myUtilityLibrary`. These declarations identify the usage of an external QVT-O definition. The main entry point of a QVT-O script is a transformation and the imported files declare libraries that can be reused. The import statement represents the link to the external files without their file extensions. Due to the access statement, the transformation gains access to the library defined in the external file.

Operation Types

The entry point to the transformation is the main() operation. The QVT-O transformation engines will look up and execute it. Besides the main() operation, QVT-O provides three different types of operations that can be declared in a transformation or a library.

- **Mapping** is the standard operation to create new elements in the transformation.
- **A Helper** is used to specify a general functionality. It has the same power as a mapping operation.
- **A Query** is a read only operation. As the name implies, its intention is to locate and access objects in a model.

QVT-O provides helper and mapping operations with a slightly different syntax but comparable power. Both are able to receive one or more input parameters and able to return one or more output parameters. Furthermore, both handle input parameters as references and are able to modify the reference elements too.

The significant difference of these two operations, is that a mapping operation caches its result for a specific set of parameter values. If the mapping is called a second time with the same parameters at a later stage, it is not executed again but the result of the first execution is returned instead.

The QVT-O specification [Obj10a] provides the following definitions to further clarify the difference between mappings and helpers:

"MappingOperation: A mapping operation is an operation implementing a mapping between one or more source model elements and one or more target model elements."
"Helper: A helper is an operation that performs a computation on one or more source objects and provides a result."

The QVT-O specification also states that it is illegal to create new objects in a helper except for sets, tuples and intermediate properties. In addition to this rule it also makes sense to use helpers to combine a sequence of mappings to be triggered and either none or only one reference of a created element be returned. Helpers can therefore be used for calculations as well as for code structuring.

All of these operation types are specified in the same way as presented in Listing 2.3. The parentheses provide a list of input model elements. Next to the parenthesis a colon is followed by the return type of the operation. Declaring multiple return types requires to separate them with a comma. If the operation does not return any value, neither the type nor the colon is used. If the parent transformation specifies more than one output model, the @ character followed by the name of an output model can be used to identify the target model an element should be created in. The @ syntax can be used to identify the target model for in-line object creation as well as shown in listing 2.3.

Listing 2.3: QVT-O Operation Declaration

```qvt-o
mapping myOperation(inputModel: Class) : Entity @ targetmodel {
    name = "name";
}

helper myOperation(inputModel: Class) : Entity {
    var e : Entity @ targetmodel = Entity { name = "name" }; 
    return e;
}

query myOperation(inputModel: Class) : Entity {
    var e : Entity = Entity { name = "name" }; 
    return e;
}
```

OCL Integration

QVT-O provides a programming language with features specific to the requirements of model transformation.

The most significant feature is the integration of the Object Constraint language (OCL) to select a set of model elements. As an example, listing 2.4 contains a statement to select all StartAction elements from a Service Effect Specification and to choose any of them. The argument of the any() operation is a boolean expression. This expression can be used to specify a subset of elements the any() operation should choose one of. The “true” keyword in this case identifies that every element can be chosen.

Listing 2.4: QVT-O OCL example

```qvt-o
seff.steps_Behaviour->select (s | soclIsTypeOf(StartAction))
    ->any(true);
```

Element and Collection Operations

The QVT-O language provides the choice to execute an operation on a single element or on each element of a provided set. This is achieved by using either the arrow operator
(->) on a set of elements or a simple dot on a single element as shown in listing 2.5. In this example, the mappingA() operation is executed for each element in the elementList. The oclIsTypeOf() operation is only applied on the single element.

Listing 2.5: QVT-O Arrow Operator

```qvt-o
elementList ->map mappingA ();
```

As an alternative for an operation call with the arrow syntax, the **forEach** construct can be used to execute a complete block of instructions for each element of a list. Listing 2.6 gives an example of how to select all EmitEventAction elements of a repository and to execute the remove operation of the repository as well as producing a log entry for each element. The parenthesis of the **forEach** construct contains an identifier to access the actual processed element within the block.

Listing 2.6: QVT-O forEach Construct

```qvt-o
pcmRepository . objectsOfType ( EmitEventAction ) ->forEach ( actionElement ){
    pcmRepository . removeElement ( actionElement );
    log ("element removed");
};
```
3. Related Work

This chapter discusses the related work in the context of this thesis. There are only a few existing research papers that combine the areas of performance prediction and event-based systems. More work has been done specifically for one of these topics in general. The following subsections discuss the most relevant representatives of these groups. Additionally, the existing approaches to integrate performance completions in the Palladio Component Model according to Woodside et al. [WW04] are presented and discussed in conjunction with the presented approach.

3.1. Component Models with Event-Based Communication

The goal of software architecture is to define a software structure and development process to ensure that all requirements, functional and extra-functional, will be met. It generally focuses on the software design and structure on a high level instead of dealing with the low-level implementation details.

One of the established architectural styles is the component-based approach that structures a software system into encapsulated parts described by interfaces between them [RH08]. Various component models with different applications and characteristics have been developed in this field. Some of them concentrate on modelling in general such as CORBA or SCA. Others add an extra focus such as performance or quality predictions as done by PCM, QImPrESS SAMM or CBML.

Most of the approaches agree to the critical performance impact of the communication pattern used in the component interactions. Nevertheless, there is no explicit support for asynchronous event-based communication yet. All of these models provide general modelling capabilities for the processing flow such as forks, joins or loops but do not provide first class entities to support asynchronous many-to-many communications.

An overview of some exemplary component models is given below. There are two groups of component models, one with a focus on the implementation and the other one with a focus on the conceptual design of software systems. Representatives of each group are mentioned and their capabilities in the field of event-based communication are described.

3.1.1. Implementation Oriented Models

Implementation oriented component models are designed with the goal to support the implementation of a software. They are often derived from previous solutions to simplify
the next product and not about the prediction of software properties in advance. For example, code generation and project organisation are some of the widespread goals of implementation oriented models.

3.1.1.1. CORBA

One of the first model specifications for distributed and object-oriented systems was CORBA. It was developed by the Object Management Group [Gro04]. The main concept of CORBA is an Object Request Broker (ORB) which can be used to manage the communication between the individual components of a software. The specification is not related to a specific programming language or platform. It provides the Interface Description Language (IDL) which is used to describe interfaces in a platform independent way. This interface description is used to generate stubs and skeletons for a specific platform to hide the distributed communication from the developer of the component itself.

The CORBA specification defines event services for asynchronous communication. These services enable push and pull based communication. With CORBA 3.0 the definition of a CORBA Component Model (CCM) has been introduced [Gro06]. It defines ports that provide interfaces to other components. The model explicitly defines Event Sources and Event Sinks as different types of ports. Furthermore, CORBA differentiates two types of event sources. The first is named emitter and can be connected to zero or one consumer. In contrast, the second type is named publisher and an arbitrary number of sinks can subscribe to such a port. CORBA explicitly distinguishes between the term connect in the first case and the term subscribe in the second case.

CORBA is a pure design oriented model with a strong focus on the interface description to generate stubs. It does not take any quality related information into account and therefore there is no tool support for any performance predictions based on CORBA. There is only a small overlap of CORBA and the approach presented in this thesis. The overlap occurs mainly in the conceptual view on event sources and sinks as well as the definition of explicit event types.

3.1.1.2. SCA

Another component model with a strong focus on inter-component communication including asynchronous invocation is the Service Component Architecture (SCA) developed by the Open SOA Collaboration [Ope10]. The foundation of industrial partners including IBM, Oracle, SAP, and Capgemini formulated the goal to develop a model for building applications with a Service Oriented Architectures (SOA).

While the specification covers a lot of modelling aspects for service component architectures, it falls short in the area of quality of service descriptions. The specifications include an explicit policy framework [Fed07], but it is more related to communication policies and reliability and does not cover any aspects of performance.

To support the event-based communication in SOAs, the foundation published an extension for their assembly model to cover Event Processing and Publish/Subscribe concepts [Fed09]. As implied by the name, the processing of the events itself and the publish/subscribe model are the main topics of the specification. While the SCA is focused on SOAs and their implementation, the specification of the events is also tightly coupled to that goal. Events are treated as a communication option between components and the focus is set more on the description of event filters and the processing of events. For example, events do not necessarily have a specific event type and neither sinks nor sources are represented as explicit model elements.

As with CORBA, SCA is more of an implementation oriented approach and does not provide performance prediction facilities as targeted in this thesis.
3.1.2. Analytical Component Models

Analytical component models are developed with the goal to analyse aspects of a software and are not necessarily required to support its implementation in any way.

3.1.2.1. QImPrESS SAMM

QImPrESS is a research project funded by the European Union with the goal to provide service orientation to critical application domains with guaranteed end-to-end quality. This includes prediction as well as verification of Quality of Services (QoS) properties. Within the project a component meta-model named SAMM has been developed [BBB+08]. This meta-model is similar to the Palladio Component Model with a couple of modifications specific to the requirements of the QImPrESS project. The meta-model also describes elements for event-based communication including support for many-to-many relationships between connected components.

The QImPrESS project also covers quality of service requirements such as system performance. Similar to the OMG performance prediction process [22], it makes use of the PCM workbench for the performance prediction. Instances of the SAMM meta-model can be transformed to instances of the PCM meta-model and then predicted and simulated by the PCM workbench.

The SAMM meta-model has a lightweight description of event related elements. It specifies a general EventPort class with a boolean flag identifying if it is acting as a source or sink. The connector class in the SAMM meta-model is not limited in the number of connected end points to support many-to-many connections. Furthermore an event port specifies the type of message it is able to mediate. If components exchange events of multiple types as for example in graphical user interfaces, then in the SAMM model separate event ports are needed to identify each event type separately.

However, at the time of writing, the event-based elements of the SAMM meta-model are not yet supported by any quality of service prediction, especially in the field of performance analysis. Furthermore, the approach presented in this thesis provides a better usability if multiple event types are exchanged between two components. Event sources and event sinks are also handled as explicit first class entities and therefore more intuitive for the architect.

3.1.2.2. CBML

The Component-Based Modelling Language (CBML), developed by Wu and Woodside and described in [WW04] is an extended version of Layered Queuing Networks (LQN) with enhanced support for software components. The main goal of CBML is to describe a software architecture and integrate different components into predefined slots. Components should provide their own, reusable performance model as an LQN which can be considered in an overall performance analysis.

CBML provides two different views on an architecture. The External View describes the general system architecture with a black box view of the components. Components are defined by in-ports and out-ports aligned to the UML2 required and provided interfaces. Additionally, the external view describes the slots to be filled with concrete components and are possible variation points in the architecture. The Internal View describes the internal processing of a component as a layered queuing network and connects the tasks and the processing resources.

As part of the CBML analysis, the LQNs of the internal views are placed in the appropriate slots of the LQN which describes the system assembly. When this is done, the completed
model is analysed by an LQN solver and the results can be reviewed to support any design decision. With this process Woodside and Wu brought in the idea of completions to automatically complete system models with different infrastructure specifications for quality prediction models [WPS02].

CBML defines an XML schema as well as a graphical notation. The latter mainly lacks in the alignment to established notations such as the UML even if it has been taken into account when the language concept was developed. Furthermore, CBML handles resource demands as CPU processing time and thread count only. For example, it is not possible to define any semaphores or other limited passive resources.

Woodside and Wu confirmed that the communication pattern used between components has a noteworthy impact on the performance of a software system. Nevertheless, CBML does not offer any support for modelling communication aspects, especially for event-based and asynchronous communication.

3.2. Performance Prediction of Event-Based Systems

Event-based communication has been of research interest in the software engineering community for years, especially in the field of quality predictions. This has led to a wide range of approaches to analyse the characteristics of event-based communications. But most of them are focused on analytical models for the event distribution infrastructure and do not consider the complete system architecture.

In the following, we present representatives for different performance prediction approaches with the support of event-based communication and discuss their focus in comparison to the approach of this thesis.

3.2.1. Sachs et al.

Sachs et al. used an approach to analyse the routing behaviour of distributed event-based systems to analyse their performance [Sac10],[KSBB08]. Based on workload analysis on the routing nodes, they derived a model for the event distribution independent from the specific routing algorithm used in the distributed event-based system. The approach uses Queuing Petri Nets (QPN) to represent and analyse of the software with appropriate tools. As QPNs are not focused on software alone and limited in their modelling of complex systems, Sachs et al. developed extensions for QPNs to target these limitations. Nevertheless, QPNs are analytical models without modelling capabilities comparable to the Unified Modelling Language.

Sachs et al. have a focus on the event distribution middleware of software systems. While their approach is comprehensive in this area, it does not provide any modelling capabilities for an overall system to examine event-based communication as one aspect in a complete architecture. Furthermore, the missing separation between resource, usage, implementation and assembly model limits the usability to adapt one of them.

3.2.2. Gorton et al.

Gorton and Liu presented an approach to use Queuing Networks for the prediction of Message Oriented Middleware systems [LG05]. They had a clear focus on the Java Enterprise platform. Gorton and Liu used a Queuing Network to model the internals of the messaging system and to analyse its specific performance properties. Internal characteristics such as filters and replications are the focus of their research as also investigated by [MH06] and [JM05] who used slightly different analytical models but with a similar goal.

As Sachs et al., they do not consider the overall system architecture nor do they provide a model oriented approach as proposed by the OMG.
3.2.3. Palladio Component Model

The Palladio component model presented in [2.3] has a design and model-based approach to performance prediction. In contrast to the other approaches presented before, it provides a model for an overall system architecture and the different performance influence factors instead of modelling only a specific part of a system with an analytical model.

There are existing studies on introducing message oriented and event-based aspects in the PCM. For example, [Heu09] worked on integrating message oriented middleware aspects into the PCM. [Hes10] extended this work with the aspects of filters and topics. Becker used the properties of message oriented middleware systems in his thesis on Coupled Transformation for QoS Enabled Component-Based Software Design [Bec08b]. Kapova et al. used a message oriented middleware example in their research on model transformation [KG09]. Happe et al. used the event-based communication as an example for their research on performance completion which is discussed in the next section [HBR+09].

However, none of the existing approaches to introduce event-based communication in the Palladio Component Model examined and improved its modelling capabilities for event-based communication and many-to-many component interactions. While also targeting this issue, the presented approach reflects the existing ones during the design and implementation of the prediction of the new event related capabilities.

3.3. Performance Completions in the PCM

Wu and Woodside introduced the idea of using completions to automatically complete a software architecture with more detailed information before the model is simulated or analysed [WPS02]. This concept has been reused in the Palladio context within a couple of research activities as presented below.

3.3.1. Performance Completion for Message Oriented Middleware

Happe et al. describe a concept of performance completions to combine high-level software architectures and performance relevant characteristics of low-level implementations of a message oriented middleware product [HBR+09]. Figure 3.1 presents an overview of their integration process. In the original software architecture, the architect is able to design the software system without any low-level details. In the mark model, he defines the options of the target infrastructure. The main transformation uses a completion library providing pre-defined transformation chunks. The appropriate chunks are selected based on the mark model options and included in the overall transformation. The resulting transformation is used to augment the original PCM model with more detailed information about the underlying infrastructure. This setup frees the architect from modelling all the platform specifics by himself. Figure 3.2 illustrates an example for a marked assembly connector in the context of inter-component communication with a messaging system in between.

![Figure 3.1.: Integrating Performance Completions (Source: Happe et al. [HBR+09])](image-url)
Christian Heupel picked up the concept of Happe et al. to model a message oriented middleware (MOM) in the Palladio Component Model [Heu09]. He implemented a QVT-R based transformation from an annotated, abstract model to a more detailed PCM model with the MOM components in place. Figure 3.3 shows the messaging completion components used by Heupel. They distinguish between sender, messaging middleware and receiver components. This model is also used in the on-going research on performance relevant characteristics of message oriented middleware.

In our approach we take this model into account for event-based communication with a message oriented middleware, especially to integrate the results of the on-going research of Happe et al. on MOM characteristics.

The completion concept used by Heupel is limited to MOM systems, especially to the messaging queues in the context of the Java Messaging Services. In contrast to this, the approach presented in this thesis provides a solution for any kind of event-based communication and is not limited to those built up with a central and JMS based middleware. The setup used by Heupel is one of the possible type of setups of such systems.

Furthermore, Happe and Heupel focused on the integration of a fixed central middleware definition into a software system. The architect can choose only from a few preconfigured characteristics of this middleware. For example peer-to-peer systems would not be possible with this approach. Furthermore, they did not introduce any changes in the PCM metamodel to support the explicit modelling of event-based communication.

3.3.2. Higher Order Transformation (HOT)

In [KG09], Kapova and Goldschmidt described an approach to use feature models as mark models and to generate a transformation which is applied to the software model. Feature models were invented by Czarnecki and Eisenecker [CE00] and are able to describe possible feature configurations in a compact way as feature trees. Kapova and Goldschmidt used the advantage of feature trees to represent a large number of possible characteristics of a system. A feature model of a specific system configuration is then used as an input to automatically adopt a general transformation to retrieve the analytical model for a performance prediction.
Kapova et al. developed a completion framework for the Palladio workbench described in \[2.3.3.4\]. Until now, the framework supports QVT-R based transformations only. Due to the advantages of QVT-O it is used for the implementation of the presented approach. An extension of the framework to support QVT-O is therefore provided as part of this thesis.

The HOT approach is focused on the transformation of a model. It neither provides explicit modelling facilities for event-based elements nor for the modelling of middleware infrastructures.

### 3.3.3. Coupled Model Transformation

As mentioned before, Becker presented an approach of coupled model transformations. This is a general approach as presented by Wu and Woodside to combine a model of the overall software system with a more detailed model of a part of the system. Becker used this approach to combine a PCM model that contains components distributed on multiple infrastructure nodes with a communication middleware model. He replaced the linking resource which represents the network connection in the original PCM model, with the detailed model of the communication middleware.

As done by Happe et al., this approach enables the architect to define an infrastructure system and to automatically add these details to a more abstract system model. Nevertheless, even Becker only focused on the transformation and did not introduce any meta-model extension to identify component connections to weave the middleware into.
4. Approach

The goal of this thesis is to provide advanced modelling capabilities for event-based communication in component-based software architectures on the one hand and the prediction of their quality attributes on the other hand.

There is no existing solution which satisfies this goal at present, but there are a select few which provide a base for parts of the solution. Therefore the approach of this thesis is to build a solution as an extension and combination using the most relevant solution available to date.

The Palladio project provides advanced capabilities for the modelling of component-based software architectures in general as well as their prediction. Furthermore, it makes use of model-driven software development techniques as well as the Eclipse platform and its modelling project. These advantages led to the decision to build a solution based on the Palladio Component Model (PCM).

Using the PCM as a foundation, the approach will consist of three contributions to reach the goal of this thesis. Figure 4.1 provides an overview of the contributions and their interplay.

Firstly, the PCM meta-model is extended in its capabilities to model event-based communication. Especially the missing support for many-to-many connections and asyn-
chronous communication is complemented. In order to build an integrated solution for the PCM, some parts of the meta-model have been refactored for more flexibility in the inter-component connection. On top of this refactoring, explicit elements to model events have been added. All of these changes have been done in alignment to existing concepts of the PCM.

The enhancement of the meta-model also took usability aspects for the architect into consideration. This is done on the semantic level by providing EventGroups for systems with a large number of different events exchanged between the same components as an example. The approach includes a graphical notation for the new elements and an extension of the graphical editors included in the PCM workbench.

The second goal being the prediction of the impact of event-based communication on the overall system is also targeted based on the PCM workbench. Built on a study on covering asynchronous communication with meta-model elements already supported by the PCM [RK09b], a component chain has been developed representing a detailed processing of event-based communication. This component chain can be processed by the existing prediction units of the PCM workbench.

To fill the gap between the new meta-model elements and those accepted by the prediction unit, a model-to-model transformation has been developed to transform the new event related elements into a set of classic elements that are able to represent them. State-of-the-art model-driven development techniques are used to implement this transformation.

The third part of the contribution supports the decision of the implemented middleware. So far the approach enables the modelling and the prediction of the event-based communication in general. To entirely support the architects design decisions regarding event-based communication, the characteristics of the specific middleware platform must be taken into account. To deal with this, the transformation weaves components of a middleware specific repository into the overall architecture. This repository contains the components and characteristics of the middleware platform it represents and the transformation takes care for their deployment.

All these contributions respect the usability for the software architect. The new modelling facilities are represented in graphical editors and the transformation is integrated in the Palladio simulation process. For the integration, an extension of the PCM completion framework is provided by this thesis to support the operational Query View Transformation language.

The next chapter describes the extended modelling capabilities in detail. This is followed by a description of the transformation to the communication component chain and the process to weave the middleware into it. The final implementation chapter describes the significant and innovative aspects of the implementation.
5. PCM Model Extension

In order to model event-based and asynchronous communication in the Palladio Component Model this approach will introduce extensions of the interaction related parts of the PCM meta-model. In detail the modelling requires the specification of events, the event-based connection between components and the creation as well as the handling of events.

The meta-model adaptations strive for two major goals. Firstly, they provide modelling capabilities for the semantics described above. Secondly, the new elements have to be compliant with the overall PCM modelling concepts.

The PCM is aligned with the component description of Szyperski et al. [Szy98]. One of their main concepts states that components are described by their interfaces as a contract of their usage. The PCM provided only one type of interface with signatures for synchronous operation calls. These signatures have zero, one or many parameters and an optional return type. This limits the modelling explicitness. For example, when modelling asynchronous event-based communication, the architect should not be allowed to model any return types.

5.1. Meta-Model Extension

The presented approach extends the interface and signature descriptions in the PCM meta-model. It introduces an abstract interface and signature description to enable more explicitness for different types of component contracts. This enables the distinction between operation and event-based interfaces as well as signatures and their divergent characteristics. Furthermore, the changes of the interface description provides the infrastructure for additional interface types already planned by other research groups. These other types include infrastructure and resource interfaces and signatures with their individual behaviour, parameter and return value characteristics.

The meta-model changes are described in more detail below.

5.1.1. Events

Interfaces describe the contract between two components. In an operational, synchronous communication, interfaces combine a set of signatures. These signatures describe operations that are required by one component and provided by another one. In the event-based communication, the contract does not describe a set of operations to be called but a set of events one component might emit and one or more components can handle. As presented
in Figure 5.1 the meta-model extension of the presented approach contains an EventGroup as a specific type of the abstract interface class. EventGroups are named entities and therefore an architect can provide a reasonable name for each of them. An EventGroup represents a contract which components can either require or provide. To describe the individual events which can be produced or consumed, the approach includes a meta-model element named EventType. This element is a specific type of Signature and the counterpart of an OperationSignature. The significant difference is that an EventType references only one Parameter to describe its characteristics but it does not describe any return values caused by the asynchronous invocation style.

Depending on the specific scenario, the performance relevant aspects of an event can vary from a single property such as a topic to a complex structure of the event content. For example, a system might be influenced only by the topic stored as a string in the event or a filter might analyse the event content in detail to decide whether any further processing is necessary. Our approach provides the flexibility to represent this variety in the meta-model. The Parameter element referenced by an EventType can either have a primitive or a complex DataType and the architect can choose the appropriate type for his scenario.

![Figure 5.1. Event Types and Groups](image)

### 5.1.2. Roles

Interfaces in general and EventGroups in event-based systems are the contracts for component interactions. Within a specific component connection each of the components act in a specific role. Both SourceRoles and SinkRoles require an EventGroup that includes the EventTypes the component emits events of. Other components provide event handlers for the EventTypes of the EventGroup.

The presented approach aligns this requirement with the general PCM concept for providing and requiring component contracts. As shown in Figure 5.2 the approach introduces an abstract ProvidedRole Element and an abstract RequiredRole element which describe the roles of a component within a component connection.

For the synchronous operational interaction, components can act in an OperationRequiredRole or an OperationProvidedRole. Both of them have to reference the same OperationInterface within a connection.

In an event-based interaction, the SourceRole is a RequiredRole to identify a component which might emit events of the types described by the referenced EventGroup. The counterpart is the SinkRole. This is a ProvidedRole to identify a component with event handlers for all EventTypes of the referenced EventGroup.

The introduced abstraction of ProvidedRoles and RequiredRoles supports the PCM’s classification of providing, requiring and complete component types. They are linked by the abstract meta-model classes InterfaceProvidingEntity and InterfaceRequiringEntity. As a subclass of these two, the InterfaceProvidingRequiredEntity and therefore also the repository components are linked with the existing operation roles and the new event roles.
5.1.3. AssemblyEventConnector

When a component developer has specified the available components to emit or to handle events, the architect defines composed structures such as systems or composite components based on these components. If there are multiple sources and sinks for the same Event-Group in one composed structure, then not every sink might subscribe to every source. Therefore, the architect has to specify the desired connections between sources and sinks.

In the existing PCM meta-model this requirement already existed for the operation calls between components. To meet the requirement, AssemblyConnectors which are aligned with the UML are provided to connect an OperationRequiredRole with an OperationProvidingRole. In this specific operational case, only one-to-one connections were allowed. In the event-based scenario, SinkRoles are able to handle events that are emitted by one or more SourceRoles and the events of one SourceRole can be handled by zero, one or many SinkRoles.

The presented approach introduces an AssemblyEventConnector to represent the connection between a sink and a source component. This AssemblyEventConnector is aligned to the design of an operational AssemblyConnector with the exception that an arbitrary number of AssemblyEventConnectors can start at the same SourceRole or end up in the same SinkRole.

As presented in Figure 5.3, the AssemblyEventConnector is a link between a SinkRole and a SourceRole. Furthermore, it has an association to the AssemblyContext of a sink component and a reference to the AssemblyContext of a source component. These contexts are required to uniquely identify the role of the component instance, especially if a repository component has been deployed multiple times. The AssemblyEventConnector is part of a ComposedStructure to identify the owner of the connector itself.

5.1.4. DelegationConnector

Designing ComposedStructures such as Systems and CompositeComponents, requires to specify the external roles and the internal processing for them. In return if an internal component requires an interface not provided within the composed structure it needs to be delegated to an external required interface.

The presented approach, introduces delegation connectors for event distributions between inner and outer roles of a composed structure. These connectors are aligned with the existing connectors to delegate operation calls between internal and external roles.

As presented in Figure 5.4 there are two different types of delegation connectors. The SourceDelegationConnector delegates emitted events from a SourceRole of an inner
component to an external SourceRole of the enclosing composed structure. In the meta-model this is represented by two links to different SourceRole elements. Furthermore, the delegation connector has a reference to the AssemblyContext representing the deployed instance of the component with the internal SourceRole. The SourceDelegationConnector itself is contained within a ComposedStructure. This should be the same ComposedStructure that the internal AssemblyContext and the SourceRoles belong to.

The SinkDelegationConnector is designed in the same way as the SourceDelegationConnector except of the direction it delegates events and the type of roles it connects. It identifies the internal component which provides the event handler offered by the composed structure to the external environment. It therefore represents a connection from the external to the internal SinkRole identified in the meta-model by two links to the SinkRole class. The other meta-model associations to the ComposedStructure and the AssemblyContext have the same meaning as the SourceDelegationConnector.

This meta-model design contains redundancy in the references between the specific delegation connectors, the ComposedStructure and the AssemblyContext. These references can be combined to references to and from the abstract DelegationConnector class. This clean-up is not crucial part of the presented approach and therefore not included in this thesis to limit the required adaptations in the PCM which need to be completed by other research and development teams.

![Figure 5.3.: AssemblyEventConnector](image)

![Figure 5.4.: EventDelegationConnectors](image)
5.1.5. EmitEventAction

From a quality prediction perspective, the software architect needs to model the static structure of the components and the processing inside the components. The latter includes the point in time when events are emitted and their quality related characteristics. According to the PCM service effect specification concept, a new action class has been designed to be placed in such service specification graphs. Figure 5.5 presents the meta-model context of this action class. The new EmitEventAction is able to emit events of a specific type identified by the referenced EventType class.

To provide the flexibility of multiple EmitEventActions for the same EventType but triggering different SourceRoles at a different point in time, the EmitEventAction references a specific SourceRole. This SourceRole is triggered when a new event is created by the action.

As described in the section covering the EventType element, the characteristics of the event are defined by the associated Parameter. Similar to the ExternalCallAction, the EmitEventAction includes VariableUsages with VariableCharacterisations to specify the characteristics of the emitted event.

As presented in Figure 5.6, the approach introduces a new abstraction for CallActions. The abstract CallAction were previously referenced by an arbitrary number of input VariableUsages and an arbitrary number of return VariableUsages. The latter VariableUsage is permitted in an asynchronous communication. Due to this, a new distinction between CallActions and CallReturnActions has been introduced. The primer one is the super class of the EmitEventAction.

5.1.6. Event Handler

When a source emits a new event, all connected sinks are informed automatically. Each sink will individually process the event. With the approach of this thesis, the architect is able to specify the quality related properties of the processing with the common ResourceDemandServiceEffectSpecification (RDSEFF) facilities.

In the meta-model, the RDSEFF is a subclass of the abstract ServiceEffectSpecification which describes the processing of the Signature it is linked with. As shown in
Figure 5.6.: Call Action Class Hierarchy

Figure 5.7.: there is no difference between an RDSEFF which is linked with an OperationSignature and one which is linked with an EventType. This is due to the concept that both are subclasses of the abstract Signature.

Figure 5.7.: RDSEFF for OperationSignature and EventTypes

5.2. Graphical Notation and Editors

Nowadays, software architects often use graphical representations of their models to communicate with stakeholders and for design purposes. The PCM workbench includes graphical editors to support a better usability of the Palladio Component Model instances. These editors encapsulate less important details and provide edit helpers to simplify the model manipulation. They offer an efficient and more intuitive way of working with the PCM models [Mar07].

As part of the presented approach, the PCM editors have been extended to provide the same usability for the new event-related elements as they already do for existing operational elements.

5.2.1. Repository Editor

The tools of the repository editor already provided creation tools for interfaces, signatures and operational roles. The new approach adds creation tools for EventGroups,
EventTypes and the event-related roles in parallel to the operational roles. The graphical representation of the corresponding elements in the repository editor uses the style of UML diagrams. Figure 5.8 presents an example of a repository with instances of the new elements. Sink and Source are common BasicComponents. The EventGroup element is an instance of the EventGroup meta-model class, identified by an icon with a green circle and a capital E inside. It includes EventTypes identified by an icon with a grey sheet and a capital E inside. The roles are represented by the connectors from the components to the EventGroup. SourceRoles are identified by connectors labelled <Emits> going from the source component to the EventGroup. SinkRoles are identified the same way but with a connector labelled <Handles>. The figure contains examples of such connectors between the sink component and the EventGroup and between the source component and the EventGroup respectively.

To enhance the usability, edit helpers are provided to automatically create initial SEFFs to represent event handlers for every EventType included in an EventGroup if a SinkRole is added to a component.

![Extended Repository Editor](image1)

**Figure 5.8.: Extended Repository Editor**

5.2.2. Composite Editor

To build assemblies with event-based interactions, an extension of the composite editor is provided. It introduces a graphical notation for SinkRoles and SourceRoles.

As shown in Figure 5.9, sources are represented by lollipop symbols with a triangle as the head of the lollipop. In the example, the Assembly_Source context contains such a role on the right to the assembly box. The notation of a SinkRole is a socket represented by a rectangle with a triangle opening able to fit in the source triangle. In the example, such a SinkRole is presented on the left side of the Assembly_Sink context.

![Example Composite Editor](image2)

**Figure 5.9.: Example Composite Editor**

One of the advantages of the PCM is its similarity with the Unified Modelling Language. To retain this similarity, the new composite elements are aligned with existing UML ele-
ments as well. While there are no explicit elements in the UML for events, the new PCM elements are taking its cue from by UML signals, AcceptEventActions and SendSignalActions ([Obj10b]). Figure 5.10 shows an example of signal producers and consumers in an UML activity diagram.

Figure 5.10.: UML Activity Diagram - Signal Actions [Obj10b]

5.2.3. SEFF Editor

The graphical editor for the service effect specifications is enhanced with a representation of the new EmitEventAction. As presented in Figure 5.11, the graphical notation is the same as for the ExternalCallAction but without the output VariableUsage compartment and with the stereotype label «EmitEventAction».

Figure 5.11.: Example SEFF Editor
6. Transformation

The meta-model described in the last section enables software architects to model their architecture with an asynchronous, many-to-many event-based communication. This model includes a high-level conceptual representation of the inter-component communication as known from the OMGs PIM ([Obj03]) specification.

To make use of the existing simulation and prediction units, the model passes two modification steps as shown in Figure 6.1. In the first step, a transformation of the high-level model with the new abstract event elements is required to match the capabilities of the simulation and prediction units. In the second step, this platform independent model is transformed to a platform specific model. This transformation step weaves a middleware model into the platform independent model in order, to take the platforms quality properties into account.

Surveying the high-level, event-based connection in the meta-model leads to a chain of processing steps involved in reality. Some of the steps are optional, while others are required. To facilitate a realistic quality prediction of the component communication, a set of intermediate components has been derived from the processing chain. Furthermore, to separate between platform independent and platform specific parts, the presented approach introduces the component chain, responsible for the overall data-flow, and a middleware model responsible to represent the middleware specific resource demands.

6.1. Intermediate Event Processing Chain

In reality, software communication at the protocol level is often implemented with a synchronous infrastructure. These systems carry out their asynchronous behaviour with intermediate components such as queues and hubs. In the presented approach the same strategy is applied in the component chain architecture mentioned before. The concept
enables the representation of the intermediate communication chain with components and processing descriptions already supported by the simulation and prediction units of the PCM workbench.

The following subsections describe this intermediate model in more detail.

### 6.1.1. Event Processing Pipeline

The pipes and filters pattern is used to connect the components in the communication chain. In this pipe, all components provide and require the same interface derived from the original EventGroup of the transformed sinks and sources. In the model, the EventGroups are replaced with OperationInterfaces. Each of these interfaces contains OperationSignatures for the EventTypes contained in the previously replaced EventGroup. These Signatures define event handler operations for the EventTypes with a single Parameter similar to the Parameter which described the data of the according EventType before. The return types of the signatures is empty while they do not need to return any values.

The OperationInterface-based communication is completely synchronous. To realize the asynchronous behaviour, ForkActions are used for the decoupling in the event distribution. ForkActions can contain an arbitrary number of sub-control-flows that are triggered without waiting for their response.

### 6.1.2. Source Transformation

Figure 6.2 presents the mapping between a high-level source element and its low-level counterparts. Sources are transformed to a set of components. Each of those components is responsible for a different concern in the event processing.

![Figure 6.2.: Source to Intermediate Model Transformation](image)

The deployment of these components depends on the communication style of the underlying platform. There are two groups of components in relation to the deployment. The first group only includes the SourcePort. An individual instance of this component is always deployed with the original source component.

The second group includes the DistributionPreparation, the EventDistribution and the EventSender. In a system with a central middleware node, there is only one central instance of these components deployed on the middleware resource container. If no such central middleware node exists, separate instances of these components are deployed with
every source as done with the source port. The latter concept applies for example to peer-to-peer systems.

In the overall simulation process a model validation step takes place before the transformation is executed. This validation ensures that an AssemblyContext and an AllocationContext exists for the source and sink components. Due to this validation, the transformation can assume the contexts of the source and sink components to be present.

6.1.2.1. Source

The source is the original component emitting the events. To model the described operational communication chain, the source is adopted and the event-based communication is replaced with synchronous operation calls. Strictly speaking, the component is adopted to synchronously call the SourcePort which takes care of any further processing.

To represent the operational communication in the PCM model, the transformation replaces the original SourceRole of the source component with an OperationRequiredRole. This OperationRequiredRole points to an OperationInterface created for the EventGroup of the replaced SourceRole. This interface specifies OperationSignatures as event handlers for each EventType included in the original EventGroup. Furthermore, the EmitEventActions in the source component’s SEFFs are replaced with ExternalCallActions. These external calls are all pointing to the new OperationRequiredRole.

To complete the replacement the characteristics of the emitted event are moved to the parameter of the external call. These characterisations are copied from the EmitEventAction to VariableCharacterisations included in a VariableUsage of the new ExternalCallAction. If not all possible VariableCharacterisations are set then the missing characterisations are completed with default values. Otherwise the downstream components would not know which characterisations to forward.

6.1.2.2. SourcePort

When a source emits a new event it has to be handed over to the middleware. Depending on the platform, there is a wrapper deployed with the source component which takes care of the processing on the client side. Such processing can include marshalling, queuing or similar actions. The intention of the SourcePort component is to represent this wrapper and include any processing and quality related demands. It is therefore it is deployed on the same system infrastructure as the source component it belongs to.

In the PCM model, the SourcePort is represented by a BasicComponent. This component contains a ProvidedRole and a RequiredRole for the OperationInterface according to the EventGroup of the original SourceRole. For each OperationSignature in the OperationInterface a RessourceDemandServiceEffectSpecification (RDSEFF) is created in the SourcePort component. The RDSEFFs contain a StartAction, a StopAction and an ExternalCallAction in between. This ExternalCallAction calls the same signature in the OperationRequiredRole as the RDSEFF is interlinked with the OperationProvidedRole. To forward all information regarding the transmitted event, the ExternalCallAction contains a VariableUsage with VariableCharacterisations for each available parameter characterisation. For example, if the parameter of the event is named “eventContent”, the specification for the STRUCTURE characterisation is “eventContent.STRUCTURE”. This is provided for all characterisations in the same way.

For the SourcePort, an AssemblyContext is created in the same ComposedStructure as the source itself. Afterwards, an AssemblyConnector is created to link the source component and the SourcePort.

To finally deploy the component, an AllocationContext is created for it in the same resource environment as the source component.
6.1.2.3. DistributionPreparation

When a new event arrives at the middleware some processing might be required at this point. This processing takes place once per event and is independent of the number of recipients interested in it. The processing can include middleware-side marshalling, security-checks, compression or others. This varies with the underlying platform. To trigger the specific resource demands, the DistributionPreparation component is located as the first process on the middleware-side component chain.

The DistributionPreparation is represented in the PCM model in the same way as the SourcePort. The only difference is the naming and the processing in the subsequent transformation. The AssemblyContext for this component is created in the same Composed-Structure as the SourcePort component. Both components are then connected with an AssemblyConnector.

In the case of a central middleware which is not running on the same infrastructure as the source component and includes the DistributionPreparation, this component is deployed on the same server as the central middleware. This is done to analyse any resulting network traffic at the right point of the communication process. To represent this in the model a new AllocationContext is created for the AssemblyContext of the DistributionPreparation on the according middleware resource container.

If no central middleware exists, the AllocationContext is created on the same Resource-Container as the source component.

6.1.2.4. EventDistribution

The event-based communication is often used in asynchronous and many-to-many relationships between components. The EventDistribution component has been introduced to represent the processing sub-path for the replication on one side and for the decoupling on the other side. Its internal processing triggers a new control flow for each downstream sink and it therefore realises the asynchronous behaviour.

The representation of the EventDistribution in the PCM model is a BasicComponent once again. It has a ProvidedRole for the OperationInterface as the previous others, but it has separate RequiredRoles for each sink which is connected to the original SourceRole. The RDSEFFs created in this BasicComponent contain a ForkAction which contains ForkedBehaviors with a Start-, Stop- and ExternalCallAction for each RequiredRole as shown in Figure 6.3. These ExternalCallActions are linked with one of the RequiredRoles and contain VariableUsages with forwarding VariableCharacterisations as described earlier in the SourcePort and DistributionPreparation components. The ForkedBehavior does not include any SynchronisationPoint and it therefore triggers the ForkedBehaviours and continues the process without any delay.

The EventDistribution component is placed in an AssemblyContext in the same ComposedStructure as the DistributionPreparation. Both of them are then connected with an AssemblyConnector. An AllocationContext for the EventDistribution is created in the same ResourceContainer. The EventDistribution component is deployed with the same logic as used for the DistributionPreparation.

6.1.2.5. EventSender

When the middleware has performed the event replication it forwards the event to the recipients. This step might include additional processing such as compression, transformation, filtering or others. The sending itself can therefore have quality related resource demands. To reflect this in the model a separate EventSender component is placed in the
chain per recipient. This EventSender triggers the demands in the platform related to this sending.

The EventSender is again represented in the PCM model in a comparable manner to the SourcePort component with a different name and further processing. An AssemblyContext and an AllocationContext for the EventSender are created in the same ComposedStructure respectively as in the same ResourceContainer as the EventDistribution before.

AssemblyConnectors represent the links between the EventDistribution and the individual EventSenders. At this point it is assured that only one AssemblyConnector is linked with one RequiredRole of the EventDistribution.

### 6.1.3. Sink Transformation

The second part of the intermediate model creation is the transformation of high-level sink elements. As with the sources, each connected sink is replaced with a set of components. All of these components are deployed once per sink and do not depend on the existence of a central middleware node in the resource environment. Furthermore, sinks that are not connected to any source are simply deleted to match the according PCM validation criteria. Figure 6.4 provides an overview on the mapping described in the following paragraphs.
6.1.3.1. EventReceiver

When an event arrives at a sink it first needs to be accepted. This might involve processing like de-marshalling or require a limited resource such as a thread pool. An EventReceiver component is placed in the communication chain and deployed with every sink component.

The representation of the EventReceiver in the PCM model is a BasicComponent which forwards the parameter characterisations of the incoming event. The BasicComponent has a ProvidedRole and a RequiredRole for the OperationInterface that replaced the EventGroup of the original SinkRole. The BasicComponent contains a RDSEFF for each Signature of this interface. All of these RDSEFFs contain a StartAction and a StopAction with an ExternalCallAction in between. This ExternalCallAction references the OperationRequiredRole and the same Signature that is handled by the actual RDSEFF. The ExternalCallAction has a VariableUsage with VariableCharacterisations to forward all possible characterisations as described earlier for the SourcePort component.

The EventReceiver is deployed along with the sink component and an AssemblyContext is created in the same ComposedStructure. The AssemblyConnector is also placed in this structure and connects the EventReceiver to the EventSender. The AllocationContext for the EventReceiver is created in the same ResourceContainer as the connected instance of the sink component.

6.1.3.2. SinkPort

Finally, the event needs to be handed over to the original sink component. Depending on the platform this might have resource demands related to this step. Even if this is often in smooth transition with the event receiving step, it is separated in this approach to not limit the flexibility for different platforms at this point. The SinkPort is a component assembled in the processing chain to represent the specific demands.

The SinkPort is created almost the same way as the EventReceiver. The component and the contained RDSEFFs are the same except for the naming and the further processing. The AssemblyContext and the AllocationContext are also created according to the sink component. Both components, the EventSender and the EventReceiver, are then connected with an AssemblyConnector.

6.1.3.3. Sink

The original sink component is modified to handle the incoming operation calls instead of the emitted events. The previous SourceRole is replaced with an OperationProvidedRole. This OperationProvidedRole is linked with the OperationInterface which has replaced the EventGroup. The existing RDSEFFs are now linked with the Signatures of the OperationInterface instead of the EventTypes. The name of the event content parameter is still the same as in the VariableCharacterisation of the original EmitEventAction in the source component and does not change in the communication chain. Due to this fact, the RDSEFFs of the sink do not require any modifications in their internal processing and are still valid for the new incoming calls.

6.2. Middleware Platform Specification

The available middleware products for event-based communication provide a wide range of event transmission architectures. This includes central message hubs, peer-to-peer systems and numerous architectures in between. The communication style and the platform used for it can have a large impact on the quality properties of the overall system. To decide on a platform and to find the right configuration for it, a software architect might need to select, configure, and test many different setups.
From a modelling point of view, the general event-based connection between components and the specific middleware used for the technical implementation are on two different levels of abstraction. To prevent unnecessary work, the architect might not want to change the high-level model for every platform he would like to test. To support the architects work in this area, the presented approach enables him to use separate middleware models without any modification of his high-level architecture. The presented approach contains a transformation that automatically weaves the additional middleware model into the component chain described above.

The separate middleware model has to provide predefined interfaces as presented in Figure 6.5. There are individual middleware interfaces for each component of the component chain. How many components are used to provide those interfaces in the middleware is up to the architect and depends on the specific middleware product. This could include individual components for each interface or only one component for all of them. Figure 6.5 contains three possible alternatives for the middleware model.

The middleware weaving is part of the transformation process developed for this approach. Figure 6.6 presents the required steps of this weaving sub-process.

In the first step a lookup for the middleware interfaces based on the naming conventions is performed. The defined interface names are:

- IMiddlewareSourcePort
- IMiddlewareDistributionPreparation
- IMiddlewareEventDistribution
- IMiddlewareSender
- IMiddlewareReceiver
- IMiddlewareSinkPort
In a second step, required roles for these interfaces are added to the appropriate components of the platform independent component chain. If this is done, an \textit{ExternalCallAction} is added as the first action of the service effect specifications of these components. The \textit{ExternalCallAction} includes a \textit{VariableUsage} and \textit{VariableCharacterisations} to forward the characteristics of the currently processed event to the middleware.

As mentioned in the introduction to this chapter and already applied during the deployment of the platform independent components, the middleware components are deployed depending on the selected resource environment. If the selected environment includes a \textit{ResourceContainer} named “Middleware”, instances of the middleware components providing the middleware interfaces for the DistributionPreparation, the EventDistribution and the EventSender are deployed centrally on this \textit{ResourceContainer}. If this central middleware container does not exist, individual instances of these components will be deployed for every source component.

With this automatic deployment concept the architect can choose between a source infrastructure deployed as singleton or once per source. From his perspective he can simply specify a middleware container or let the transformation take care of the peer-to-peer distribution.

As soon as \textit{AssemblyContexts} are in place for all specified middleware components, the transformation creates the \textit{AssemblyConnectors} to couple the platform independent components with the appropriate middleware components.

The specification of the middleware internals is completely up to the architect and can be done with the classic PCM model elements. For example, the software architect can model simple resource demands, control flows or passive resources. This middleware model can include research results as mentioned by Happe \cite{HBR*09} and others.

In the middleware model, the usage of event-based communication is not prevented but it will not be transformed during the process. At the end, this will lead to an error in the simulation and prediction units as they do not support these elements.

\subsection{Modelling System Characteristics}

The separation between the platform specific meta-model and the high-level software architecture was designed with respect to the modelling of event-based system characteristics. With this differentiation a distinction between high-level and low-level platform characteristics has been reached. The former models the data-flow while the latter models the required resource demands of the underlying infrastructure.
The middleware model describes the specific characteristics and resource demands of the middleware product. Most of them are transparent to the transformation described in this chapter. Nevertheless, there are a couple of middleware features directly targeted by the transformation. A feature tree, as described in [CE00], is used to summarise these characteristics. Figure 6.7 provides a visualisation of this tree. As illustrated, the transformation is capable to work with different types of middleware architectures. This includes single-system solutions as found in graphical user interfaces, embedded systems or smaller solutions. Furthermore, systems with a central communication server and those with peer-to-peer concept are both supported. As described by the interface sub-tree, middleware interfaces are treated as optional. For example, if a middleware does not perform any distribution preparation the IMiddlewareDistributionPreparation interface must not be in place. As mentioned before, each interface can be provided by a separate component or as part of a more complex unit. The third sub-tree describes the capability to handle sources that are connected with multiple sinks as well as sinks that are connected with multiple sources.

To prove this approach of separating all relevant system characteristics, the following list contains a modelling description of the relevant messaging patterns of Happe et al. ([HBR+09], page 9) and the topics and filters analysed by Hessel [Hes10].

**Point-to-Point Channel**

The point-to-point channel is used in peer-to-peer systems. It describes the direct communication between the participants without a central communication node. The important point in this case is the more complex middleware facilities deployed with the source component. The event distribution as well as the decoupling from the process of the source has to be handled on each source infrastructure individually. Furthermore, each source, or at least the middleware deployed within the source, knows the sinks interested to its events.

To model the usage of such point-to-point channels in the presented approach, there should be no central middleware node present in the resource environment. As a result, all middleware components from the SourcePort up to the Sender are deployed once for each source on the same resource container as the source itself.
**PublishSubscribeChannel**

The publish-subscribe channel is often referred to as the opposite of the point-to-point channel. A central communication middleware is deployed for such a system. The sinks subscribe to specific events and the middleware forwards the incoming events from the publishers to these subscribers.

To model such a publish-subscribe channel, a central **ResourceContainer** named “Middleware” should be present in the selected resource environment. The transformation then takes care for the central instantiation and deployment of the middleware components.

**TransactionalClient**

A transactional client characteristic identifies that either all events of a set are deployed or none of them. Such a system requires synchronisation and roll-back mechanisms.

The resource demands for failures and roll backs should be modelled in the appropriate middleware components. Depending on the specific scenario, some more infrastructure such as call backs or others might be necessary for such a system. There is an ongoing investigation to improve the PCM capabilities for reliability and recovery. With the presented approach they can also be applied to the middleware model as soon as they are available.

**GuaranteedDelivery**

In contrast to the transactional client, guaranteed delivery always ensures the delivery. Therefore no roll-back would be required and the asynchronism is not limited.

The guaranteed delivery is implemented by queues and redundancy. The PCM already provides invocation retries, delays and similar infrastructure that can be used to model the appropriate resource demands in the middleware components.

**CompetingConsumers**

Competing customers distinguish themselves by the competitive need of a specific resource. For example, there is limited thread pool for the event delivery or a lock has to be acquired for the sink specific event delivery.

The PCM meta-model includes limited passive resources. Such a resource can be used to limit the number of concurrently active sender, receiver or sink ports. Which of them to use depends on the specific middleware product and the point in process where the competition takes place.

**SelectiveConsumer**

Selective consumers describe consumers that are picky about the events they accept. For example, this can regard specific event characteristics or the percentage of accepted events.

Selective consumers can be modelled using **BranchActions** in the RDSEFF which describes the event handler of a sink. Depending on the preferences of selective consumers, either a **ProbabilisticBranchTransition** for a percentage-based or a **GuardedBranchTransition** for an event-property-based acceptance can be used.
Topics and Filters

Some event-based systems, primarily in the field of message oriented middleware, provide a feature named topics and filters. Topics can be used to specify the events that a sink will receive. Every sink can specify the topics it is interested in. The events are marked with one or more topics and a sink will receive only those it has registered for. Filters can be used for the same purpose but provide more flexibility. They can be applied not only on topics but also on all attributes of an event.

In general, the client specifies the filter or topic it is interested in and publishes it to the middleware. The middleware takes care for the decision of whether to deliver the event to the client or not.

To provide the flexibility required to model different implementations of the topic and filter feature, the architect can choose between multiple modelling alternatives depending on the system he is working on.

If the system uses different events to represent the topics, the architect can use Event-Groups and EventTypes to make them explicit and connect the components with sinks and sources that reference the appropriate EventGroups.

If the system uses explicit topics in the events, the architect also can make them explicit in the model. He can therefore use the parameter element of the EventType and one of its characterisations as the topic. To model the data flow decision, a BranchAction in the sinks RDSEFF can be used to check if the topic is of interest and the event will be processed.

If a filter is applied then its resource demand depends on the specific platform. For example a message oriented middleware might process the filtering on a central middleware infrastructure and therefore requires hard disc access for the persisted filters. This is completely different to a peer-to-peer middleware with filters applied in the sink component. Both of these systems can be modelled by adding the required resource demands either directly in the component implementing the appropriate middleware interface or by adding it to the sink component.

This solution only reflects the resource demands on the infrastructure nodes. It excludes unnecessary network demands in the case that filters are already applied during the event distribution. The distribution of the filter definition from the sink to the event distribution on the middleware node is part of future research.
7. Implementation

This thesis provides a ready-to-use tool for the presented approach. This chapter discusses the implementation details of the tool and focuses on the parts implemented with new state-of-the-art technologies which are not widely-used at the time of writing. These implementations are realised with model-driven development languages, facilities of the Eclipse Modelling Project and the integration in the latest version of the Palladio workbench.

The first part of this chapter discusses the implementation of the transformation. The operational model-to-model transformation language QVT-O has been applied in this case. Up to now, there are only a limited number of case studies and best practices available in this area. As a contribution in this field, the following section includes the challenges solved and the solutions built in this thesis.

In the second part of this chapter, the modifications of the PCM workbench are presented. These contributions are the new PCM meta-model, the graphical editors and the integration in the completion framework.

7.1. QVT-O Transformation

Nowadays, many transformation languages are available which could be used to implement the transformation of the presented approach. The operational version, QVT-O, of the Query View Transformation Language was chosen as it is one of the most evolved standards. The reason for choosing QVT-O is its better structurability and maintainability (2.4.2.1). QVT-O’s steep learning curve for developers who are well versed in modern programming languages was another quality which supported this decision. There are also existing experiences with QVT-O from other researches in the context of the Palladio project.

7.1.1. Transformation Design

The implementation of the transformation has been designed with the aim to limit the complexity of the transformation. This includes the development of libraries to handle repetitive tasks during the model modification.

As presented in Figure 7.1, the high-level transformation process is partitioned into four steps. Firstly, Operation Interfaces for all Event Groups are created with Signatures that describe event handlers for the original Event Types. These interfaces are stored in
a global registry to simplify their accessibility in the downstream process. In the second step, the selected resource environment model is checked for a `ResourceContainer` named “Middleware”. If it is present, a centrally installed communication middleware should be used. If the middleware repository provides the related middleware interfaces, instances of the appropriate providing components are deployed on this central middleware container. The most comprehensive part is the transformation of the `SourceRoles` in the next step. Each `SourceRole` is used as a starting point to build-up the according chain of components. As a final step, the prepared model is cleaned-up from any event related elements which are not supported by the quality prediction.

![Figure 7.1.: Transformation High-Level Implementation](image)

Besides this high-level transformation process, a set of QVT-O libraries has been designed and implemented to structure the transformation. As illustrated in Figure 7.2, the libraries are partitioned into four packages. The packages are represented by grey boxes with dashed borders. They are logical packages because QVT-O does not provide any infrastructure to indicate packages. The white boxes inside the packages represent the individual files containing the QVT-O libraries and main QVT-O transformation respectively. The dashed lines between the packages and files indicate the relationships between them. The direction of information is indicated by the arrowhead.

![Figure 7.2.: Transformation Library Structure](image)

**Main Transformation**

The Main Transformation package includes only one QVT-O file with all the high-level operations. Unlike the other packages, the transformation script in this package does not contain a QVT-O library but a transformation of the main operation as the entry point. The parameters of the main operation describe the source and target models of the transformation process. This set of models is aligned with those provided by the PCM workflow.
In addition to the high-level operations, the main transformation file includes all operations that need a direct access to the input and output models. From a design perspective, some of these operations should be externalised in separate libraries, but this is not possible due to a limitation of QVT-O. Even if allowed by the syntax, models cannot be passed as parameters to operations which are not located in the main transformation script. Otherwise the import of the external file will be broken.

The four main steps of the transformation presented in Figure 7.1 are represented by four helper operations in the main transformation file as shown in Listing 7.1. Code comments have been removed in the listing for a more compact presentation. The main() operation represents the transformation entry point that manages the high-level process. The other operations encapsulate the processing of the appropriate step.

Listing 7.1: Operations of the Main Transformation Process

```java
main() {
    createOperationInterfaces();
    setupCentralMiddleware(middlewareRepository);
    Finder_findAllSourceRoles(pcmAllocation)->forEach(sourceRole){
        processSourceRole(sourceRole);
    };
    removeEventModelElements();
}
	house helper createOperationInterfaces() {...}
thouse helper processSourceRole(sourceRole : SourceRole){...}
thouse helper setupCentralMiddleware(repository : PCMREP){...}
thouse helper removeEventModelElements(){...}
```

The main process already makes use of a finder operation. These are queries which are able to find elements in the source models and therefore need a direct access to them. Listing 7.2 presents the available finder operations. Unlike the other operations, the finders are prefixed as "Finder_" and the models are provided as input parameters. This is done to support the externalisation in a separate library as soon as it is supported by the underlying QVT-O implementation. Furthermore, the finder operations are all implemented as queries and the models are marked as read-only to emphasise the read-only character of the operations.

Listing 7.2: Finder Operations

```java
query Finder_findAllEventGroups(
    in allocationModel : PCMALLOC)
    : Set(EventGroup){...}
query Finder_findAllSourceRoles(
    in allocationModel : PCMALLOC)
    : Set(SourceRole){...}
query Finder_findAssemblyEventConnectors(
    in sourceRole : SourceRole,
    in allocationModel : PCMALLOC)
    : Set(AssemblyEventConnector){...}
query Finder_findSystem(
    in allocationModel : PCMALLOC)
    : System {...}
```
query Finder_findAllocation(
    in allocationModel : PCMALLOC
    : Allocation {...})
query Finder_findAllocation(
    in assemblyContext : AssemblyContext,
    in allocationModel : PCMALLOC
    : Allocation {...})
query Finder_findResourceContainer(
    in assemblyContext : AssemblyContext,
    in allocationModel : PCMALLOC
    : ResourceContainer {...})
query Finder_findMiddlewareContainer(
    in allocationModel : PCMALLOC
    : ResourceContainer {...})
query Finder_findOperationProvidedRole(
    in interfaceName : String,
    in repository : PCMREP
    : OperationProvidedRole {...})

The remaining operations are all prefixed as "Transformation_" and located in the main transformation file while they directly or indirectly create new elements in a specific target model.

Communication Components

The Communication Components package includes individual QVT-O libraries for the components specific to the presented approach. There are two groups of libraries within the package: The Source and Sink libraries adopt the original components and the remaining ones create the new intermediate components.

The operations in the Source library take care of the adaptation of the original source component. The helper Source_transformEmitEventActions() is triggered to adapt a source component and internally change the original EmitEventActions to an ExternalCallActions. On the opposite side of the communication, the Sink library provides helpers to adopt an original Sink component. This includes the creation of an OperationProvidedRole according to a specific SinkRole. Furthermore, the SEFFs which previously acted as handler for a specific EventType are now linked to the signatures of the OperationProvidedRole’s interface. The library includes one mapping operation named Sink_createSinkOperationProvidedRole(). At this point, the caching characteristic of mappings is used to ensure that only one OperationProvidedRole is created for a specific SinkRole even if it is connected to multiple sources.

As mentioned before, the remaining libraries in the Communication Components package are all responsible for a specific intermediate component. Except of the EventDistribution, they are all implemented with the same concept presented in Figure 7.3. They create a BasicComponent according to the pipe-and-filter pattern with an OperationRequiredRole and an OperationProvidedRole according to the OperationInterface which replaces the EventGroup of the currently transformed SourceRole. Each BasicComponent is named specifically according to the appropriate intermediate component and includes a RDSEFF which forwards the characteristics of the transferred event. The new component is deployed in a ComposedStructure and a ResourceContainer previously provided as parameters. In the last step, a connector is created to link the new component with the predecessor component in the communication chain.
7.1. QVT-O Transformation

In contrast to the other libraries, the library for the EventDistribution component does not create any OperationRequiredRole and a ForkAction is added to the internal RDSEFFs but without any ForkedBehaviour. Individual instances of both of these ForkActions, ExternalCallActions and OperationRequiredRoles are created for every connected sink in the downstream process.

Common Utilities

The libraries in the Common Utilities package include operations for general modifications of a PCM model. Primarily, these are operations to create or to manipulate model elements in general and are less specific to the presented approach.

The InterfaceUtil, SEFFUtil and VariableUtil libraries provide operations specific to the model elements mentioned in the library names. The Commons library includes operations for other elements without a reasonable amount of operations to require individual libraries.

Registries

The Registries package includes operations to support a global and simplified lookup of specific model elements. These registries provide capabilities with an advanced usability over the caching characteristic of QVT-O mapping operations. For example, the OperationSignature that substitutes a specific EventType of an EventGroup is accessible via the OperationSignatureRegistry.

7.1.2. QVT-O Experience and Best-Practices

QVT-O is still a young transformation language. In comparison to well-established programming languages such as JAVA or C#, there is still a lack of experience and best-practices of how to use the flexible language concepts. The following subsections provide an overview of the concepts applied in the implementation of this approach.

7.1.2.1. Mapping vs. Helper

Mapping and helper operations provide the same power in the creation and modification of model elements. The significant difference between these two resides in their execution. While a helper is always executed when it is called, the mapping is only executed once for a unique set of parameters. If a mapping is called for the first time, the result is stored subject to the provided parameter values. For each further call with the same set of parameter values, the result of the first execution is returned instead of another mapping execution.

Helpers and mappings can still be used for the same applications and there is no better solution in general. For example, a registry can be used for a helper to return an already created element and a mapping can be forced to be executed for every call by providing an additional artificial parameter that changes for every call. Nevertheless, depending on
the specific scenario, either a mapping or a helper operation is a better fit for the specific intention.

In the presented implementation, helpers are used as the default operation and mappings are only used where their specific caching characteristics are explicitly required. This decision is based on the fact, that most of the operations in the libraries are for a general purpose and there are a numerous elements in a PCM model that are created with the same properties. For example, a `StartAction` which should be added to an `RDSEFF` is always created with the same properties. Adding the `RDSEFF` as a parameter or context to the operation would mean an additional and unnecessary parameter which inflates the operations signature. Especially in cases which would require to forward the context through a chain of operations just to create a new element and prevent mapping from returning a cached result. In the `StartAction` example, it might be required to send the `SourceRole` context through two operations up to the creation of the `StartAction` without any semantic need for it.

7.1.2.2. Main Transformation Dependency

Within the transformation process, there are multiple steps which need access to the models handed over to the transformation. This could be either to lookup specific elements (i.e. all `SourceRoles`) or to define a specific target model with the `@` operator to create a new element. In the first case, it might be reasonable to create a QVT-O library which encapsulates all these finder operations. The latter can be necessary in every library which creates new elements.

From the syntactical specification of QVT-O, there is no limitation to submit a model as a parameter to an operation, irrespective if it provides write or read-only access. Nevertheless, the Eclipse QVT-O implementation accepts this during the compilation of the library itself, but as soon as it is imported into a transformation or another library, the Eclipse implementation states the import as invalid due to an invalid library.

To handle this limitation, operations that require direct access to the complete model are placed in the main transformation file. If one of these methods is only indirectly accessed by the main transformation on a lower level in the call stack, all operations involved in between are also placed in the main transformation file. This is done to prevent any permitted access between packages and circular dependencies in the architecture of the transformation libraries.

7.1.2.3. Code Documentation

Nowadays, most of the established high-level programming languages come with guidelines and infrastructures for documentation. One of the established formats is the JavaDoc style invented by the JAVA platform and community. In QVT-O, code comments are possible but there are neither any best practices for code documentation nor a facility to generate documentation out of it.

To bypass this drawback, the JavaDoc format is used for the documentation of the transformation operations. Even if there is no tooling available to generate a documentation out of it, a great many of today’s developers are accustomed to read and understand this format.

7.1.2.4. Namespace

QVT-O does not provide any kind of namespaces to identify operations that are in the same context. Operations with the same name are only allowed when they are specified
for different element types or require a different set of input parameters (i.e. Signature::doSomething() and Interface::doSomething()). When two libraries provide operations with the same name, the same context and the same parameter set, there is no possibility to explicitly specify which one to use.

In the implementation of the presented approach, operations with the same context and parameter set are needed in different libraries. To deal with this, the operations of a library are all prefixed with the name of the library followed by an underscore. For example, the create method in the SourcePort library is named SourcePort_create() and the one for the SinkPort is named SinkPort_create().

7.1.2.5. Init, Population, End

QVT-O mappings define the code sections init, population and end. The init part is executed before the result object of the mapping is initialized. Population is the default section and the end section is executed when the mapping is finished. Init and end are optional and the population section is implicit even if the section is not marked. To promote a better code readability and explicit semantic of the code sections, init and end sections are used whenever processing is necessary before or after the population of the new object. The explicit marker for the population section is not used. It is implicitly defined as the other sections are marked if they exist. In addition to promote a clean population section, the calculation of attribute values is done within an init section before they are applied in the population. Listing 7.3 shows a mapping which defines two AbstractAction elements in the initialisation. When this occurs, the attributes of the resulting ExternalCallAction are set in the main part. This is the implicit population section. The last part of the mapping, the end-section, contains an expression to register the resulting action in the ResourceDemandingSEFF element.

Listing 7.3: QVT-O Mapping Example

```plaintext
mapping addExternalCallActionToSeff(
  inout seff : ResourceDemandingSEFF,
  signature : OperationSignature,
  requiredRole : OperationRequiredRole)
  : ExternalCallAction {
  init {
    // get the actions to place the new action in between
    var firstAction : AbstractAction = seff.steps_Behaviour
      ->select(s | s.oclIsTypeOf(StartAction))->any(true);
    var secondStepAction : AbstractAction = firstAction.successor_AbstractAction;
  }

  // set all ExternalCallAction fields
  entityName := 'emitEvent';
  predecessor_AbstractAction := firstAction;
  successor_AbstractAction := secondStepAction;
  calledService_ExternalService := signature;
  role_ExternalService := requiredRole;

  end {
    // add the ExternalCallAction to the seff
    seff.steps_Behaviour += Set{result};
  }
```
7.1.2.6. Limitations of the Eclipse QVT-O Implementation

The Eclipse Modelling Project provides an implementation of the QVT-O specification. At the time of writing, the Palladio workbench is based on the Eclipse Galileo release with a QVT-O implementation which supports only an incomplete set of the language specification. The support will be extended in the new Eclipse Helios release, but this is not supported by the Palladio workbench as yet.

Dictionary Datatype

The Eclipse Galileo release does not implement the dictionary data type of the QVT-O specification. This data type provides a map with key value pairs of data elements identified by an unique key. For example, this might be used in transformations as a registry to store and find elements. As a workaround, one can use a construct of a set of tuple data types. 7.4 presents a QVT-O code example with an element registry and operations to access it. Defined as a property, the registry data object is accessible in a global context.

Listing 7.4: QVT-O Registry Code

```qvt-o
// define the registry
property elementRegistry :
    Set ( Tuple ( key : String , element : Object ) ) = Set { };

// get a signature from the registry
elementRegistry -> selectOne ( s | s . key = registryKey ). element ;

// store a new entry in the registry
var entry : Tuple ( key : String , element : Object ) = Tuple {
    key = registryKey ,
    element = result
};

elementRegistry += Set { entry };
```

If-Then-Else

One of the advantages of QVT-O in comparison to QVT-R is the support of control flow definitions with more branches than a binary if-then-else. The QVT-O specification contains a language construct named `elif`. This construct is not included in the QVT-O implementation of the Eclipse Galileo release. As a workaround, either nested or back-to-back binary if-then-else can be used as with QVT-R.

7.2. Integration in the Palladio Workbench

In addition to the sophisticated meta-model, the workbench is the major advantage of the Palladio project compared with other approaches in the field of performance prediction of component-based software architectures. This thesis claims to not only provide a ready-to-use and validate implementation of the presented approach but to provide it as a smooth integration in the existing Palladio workbench. The goal of the workbench is, that the architect should not need additional skills other than those already required to use the PCM and that the complexity of the transformation should be transparent for the user.
To implement this, the extension of the meta-model is done in its core and the existing graphical editors are adopted instead of introducing new ones specific to the event-based communication. Furthermore, the transformation of the new elements is integrated in the SimuCom workflow and the architect only has to activate the transformation and to select the middleware repository to be weaved into the model.

The following subsections give a brief description of the implementation of the meta-model extension and details about the extension of the PCM completion framework.

### 7.2.1. Meta-Model Implementation

The PCM meta-model has been designed with the IBM Rational Software Architect and transmitted to the Ecore infrastructure provided by the Eclipse Modelling Project (EMP). The latter provides facilities for the generation of data-models, edit libraries and basic editors. They are all generated as Java Code and based on the Eclipse platform.

The meta-model extensions of the presented approach are modelled as part of the original PCM meta-model and deployed with the same tool chain.

The graphical editors are also implemented based on facilities of the EMP. The Palladio project makes use of the Graphical Modelling Framework (GMF) in this area. The tool and graph-models of the Composite, SEFF and Repository editors have been modified to extend the graphical editors of the workbench.

Firstly, the SEFF and Composite editor models have been updated to the current GMF version to be able to generate the editor base. Following this, the new graphical elements have been added to the graph-models. Creation tools for the new elements have been added to the tool-model. In addition, the map-model has been used to wire the new model elements with the new graphical elements.

After the gen-model has been generated based on the previously modified editor models, the code for the graphical editors has been generated.

As a last step, the generated code was modified to accept the new separation between operational and event-related roles, interfaces and signatures. Furthermore, the same usability functions were added for the new elements as already available for the existing model elements. This includes creation tools in the appropriate tool palettes and the modifications described below.

**Repository Editor**

An edit helper was created to automatically add valid RDSEFFs with a StartAction and a StopAction as event handlers for each EventType of the referenced EventGroup, if a new SinkRole is drawn between a component and an EventGroup.

The properties editor for roles has been extended to distinguish between operational and event-related elements and to provide only the appropriate parameter and return values.

**SEFF Editor**

In the SEFF diagram editor, an edit helper was added for the creation of an EmitEventAction, that opens a dialogue to select the EventType that should be trigger. This dialogue offers only the EventTypes available in the EventGroups of the SourceRoles defined by the actual component.
7. Implementation

Composite Editor

The new graphical notation of the SourceRoles and SinkRoles is added to the visualisation of the composite editor. Due to GMF, custom drawing facilities for the appropriate model elements were required.

The edit helpers for the creation of AssemblyConnectors have been improved to connect only SourceRoles with SinkRoles and OperationRequiredInterfaces with OperationProvidedInterfaces.

7.2.2. Integration in the Completion Framework

The PCM completion framework is integrated in the PCM workbench as part of the simulation workflow and reflected in its configuration. It already supports different types of transformations but they are all limited to QVT-R.

To reuse this framework on the one hand and to gain the advanced capabilities of QVT-O on the other hand, the framework has been extended with the support of QVT-O transformations as an additional transformation language. The modifications provide enhancements of the configuration options provided to the user and the internal processing during the workflow.

7.2.2.1. Configuration

The configuration is extended to provide the user with an option to select a QVT-O based completion as shown in Figure 7.4. To achieve this, a new transformation type named PLAINQVTO has been registered in the corresponding TransformationType enumeration class. As shown in Figure 7.5, the depending classes are adopted to handle this new type. The presentation in the UI is handled by the EditCompletionsDialog. For the existing completion types, the framework already included interface facilities to present form fields to the user to ask him for a set of files. These facilities have been reused to request the main transformation script and which middleware repository to use.

![Figure 7.4: Screenshot Completion Type Selection](image)

7.2.2.2. Completion Execution

For the QVT-O based transformation a complete new Job Builder has been added to the completion framework. Figure 7.6 presents the new PlainQVTOCompletionBuilder within its context. This builder makes use of a QVTOTransformationJob which already
Figure 7.5.: Completion Configuration Extension

Figure 7.6.: Completion Execution Extension

7.2.2.3. Working Copy Job

The completion framework makes use of the PCM blackboard infrastructure to store the PCM model to be processed. This led to problems with absolute references between...
multiple models. To prevent these problems, a new job was created and added to the PCM simulation workflow. The workflow already created a temporary Eclipse project during the simulation run. The new job creates a copy of all input models within this project and the subsequent workflow is adopted to use this working copy instead of the original files.
8. Evaluation

The goal of the thesis is an improved modelling and simulation of event-based communication in component-based software architectures. To prove its applicability and its improvement, the new approach and the previously available Palladio version are both applied on a real software system. The new approach and the advantages it provides are judged in comparison to the existing Palladio approach.

The key criteria used in the evaluation are: the required modelling effort, the expressiveness of the resulting model, the prediction quality and the support in the evaluation of different solutions to realise the event-based communication. They are also used in the Goal-Question-Metric (GQM) plan which was developed to structure the evaluation process according to Basili et al. [BCR94].

As a case study, an information and monitoring system installed in the public traffic system in the city of Cambridge is analysed. The software was provided by the University of Cambridge with recorded data from the real traffic system. Furthermore, there is a previous case study [RK09b] on this system performed with the previous version of the Palladio tools. In this case study, workarounds had been applied to simulate the event-based communication. The results of this case study are used to assess the results of the new case study based on the approach of this thesis.

While there are existing and ongoing researches on the impact of specific platform characteristics in the field of event-based systems, the presented approach assumes the correctness of their results. This thesis has already presented how to integrate the results of these researches as part of the middleware repository. The same applies for new insights which will be identified in the future.

The developed GQM plan, an introduction to the system analysed in the case study, a description of the evaluation procedure as well as the results will now be presented.

8.1. Goal Question Metric Plan

A software prediction process includes a wide range of tasks reaching from system analysis to the interpretation of the simulation results. In order to focus on the crucial criteria of the presented approach, a Goal Question Metric plan, as proposed by Basili et al. [BCR94], has been developed to judge the quality of the approach. The concept of the GQM plan is to specify the goal of the presented approach. The plan includes questions about related...
characteristics of the approach derived from this goal. On the third level, the GQM plan defines metrics to answer these questions.

Table 8.1 presents the GQM plan developed for this evaluation. The overall goal is the improvement of the modelling and simulation capabilities for event-based communication in the PCM from the software architects viewpoint. The capabilities are evaluated using four specific questions. While the improvement is always evaluated in comparison to the existing version of the PCM, the metrics for the questions are surveyed in comparison to the existing PCM capabilities, especially with regard to the first case study on the Cambridge TIME system.

The first question in the GQM plan is regards to the modelling capabilities and the expressiveness of the model itself. While this is a subjective question, a clear answer to the common explicitness of the new model elements would require an empirical study with a heterogeneous group of software architects. Such a study would exceed the time constraints of a thesis. Nevertheless, to provide a reasonable answers to this question, metrics have been chosen that cover the existence of explicit elements for event-based communication and their clear purpose within a model.

The second question targets the effort to model an event-based communication in the Palladio workbench. The selected metrics focus on the number of required elements that need to be abstracted from the individual velocity of the creating architect.

The next question covers the quality of the simulation result. While the new approach should simplify the modelling, the simulation results should provide at least the same quality as the first case study. The applied metrics compare the new simulation results with the measurements of the real system as well as the existing simulation results.

The last question concerns the capabilities to extend the model to predict additional systems and setups. The metrics used to answer this question include the number of required steps to change the middleware configuration and to change the middleware at all. Furthermore, the metrics of required steps to connect a new sink or source, as known from question two, are also considered for question four.

8.2. TIME System

The TIME (Transport Information Monitoring Environment) project at the University of Cambridge has developed a traffic monitoring and prediction system that is installed in the public traffic system of the city of Cambridge [BBE+08]. The system monitors the location of buses and the status of traffic lights to harness them for traffic flow optimization and the calculation of travelling and arrival times.

An event-based middleware called SBUS (Stream BUS) was developed as part of the TIME project [BBE+08]. SBUS was also developed at the University of Cambridge specifically for applications such as the TIME system. Its major characteristics are the peer-to-peer architecture and the support for a wide range of environments. In the TIME scenario, the architecture included embedded systems with GPS sensors in the buses as well as server-side applications for the storage of locations and prediction tasks. Figure 8.1 provides an overview of the components involved in the TIME scenario.

Bus Location Provider (the ”ACIS component”)

The bus location provider is a component installed on the buses and uses GPS sensors to record the location of a bus. As soon as the location changes, the information is transmitted to the monitoring system using a proprietary radio network and triggers the
### Table 8.1.: Goal Question Metric Plan for the Evaluation

<table>
<thead>
<tr>
<th>Goal</th>
<th>Purpose</th>
<th>Quality Issue</th>
<th>Process/Product</th>
<th>Viewpoint</th>
<th>Question 1</th>
<th>Is the event-based communication explicit in the software model?</th>
<th>Metrics</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Improve</td>
<td>the explicitness, effort &amp; adaptability</td>
<td>of modelling event-based communication in the PCM</td>
<td>from an architects perspective</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Question 2</td>
<td>Is the modelling effort reduced by the new approach?</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Is the simulation results acceptable?</td>
<td>Are the simulation results acceptable?</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Question 4</td>
<td>Is the testing of different middleware settings/setups simplified?</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Metrics</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>• Coverage of event-based communication with explicit model elements</td>
<td>• Deviation from existing measurements</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>• Expressiveness of elements in a model that represent an event-based communication.</td>
<td>• Deviation from the simulation in the first case study</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Metrics</td>
<td>• Number of steps to connect an additional sink</td>
<td>• Number of steps to change middleware parameters</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>• Number of steps to connect an additional source</td>
<td>• Minimum required steps to change the middleware</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>• Number of steps to connect an additional sink</td>
<td>• Number of steps to connect an additional source</td>
<td></td>
</tr>
</tbody>
</table>
ACIS component to emit an event concerning the location change. It is possible to install multiple ACIS components within the same environment. This can be done to balance the load on the ACIS node or to structure the system by installing separate ACIS nodes per bus line for example.

**Location Storage**

The location storage is able to maintain locations consisting of a name/location pair coupled with a timestamp. The timestamp is used to store only the latest position for a specific name. The location storage is suitable to consume events with location information and is therefore able to handle the events emitted by the bus location provider. Using a central database as a back-end, it is possible to deploy multiple instances of the Location Storage component in a system.

**Traffic Light Status Reporter (the "SCOOT" component)**

The function of the SCOOT component is to control the traffic lights. Based on the information regarding the status of each traffic light and the information of vehicle detecting induction loops installed in the road, the SCOOT component is able to optimise the use of the road network. Each status change of a traffic light triggers the SCOOT component to emit a corresponding event. There are two types of events. One type indicates that a traffic light changed to red, the other indicates a change to green. It is possible to install multiple SCOOT nodes in a system. This might be used to run individual instances per city district for example.

**Proximity Detection**

The proximity detection consumes an event stream from a SCOOT component to records the status change of traffic lights. These events are collated with a call back to the SCOOT component for further information and takes the bus locations into account to determine which buses are nearby and may be affected by a red light.

The communication between the components presented above is realised based on the SBUS middleware. A peer-to-peer event-based communication without a central middleware hub is used between the SCOOT and the Proximity Detector as well as between the ACIS and the Location Storage components. Each of these components are therefore deployed with an SBUS wrapper which regulates the necessary, asynchronous transmission of the events.

The callback from the Proximity Detection to the SCOOT component is carried out with an RPC call. The RPC infrastructure is also provided by the SBUS middleware but this is done in a traditional synchronous style. There is no direct communication between the Proximity Detection and the Location Storage but both use a direct access path to a central database which holds the location data.

The challenging point of this scenario is not only the peer-to-peer event-based communication but also the fact that multiple instances of the components can be installed in a distributed environment and operated by different parties.
8.3. Evaluation Process

To determine the metrics for the presented GQM plan, a structured evaluation process has been designed as illustrated in Figure 8.2.

![Evaluation Process Diagram]

Figure 8.2.: Evaluation Process

Reference Case Study

As a first step, the existing case study is reviewed to understand the different setups that have been analysed. For this case study, servers in a laboratory at the Karlsruhe Institute of Technology (KIT) have been set up to replay the scenario recorded in the city of Cambridge. For the evaluation in this thesis the existing Palladio model and the documented results are used for the assessment.

New Case Study

In a second step, the TIME system is modelled from scratch with the new modelling capabilities. This includes the different scenarios used in the first case study to be able to compare the required analytical efforts.

Comparison

The next step of the evaluation process is the comparison of the two case studies. This is done with a focus on the resulting models, the effort to create them and the operability of the simulation. The models are reviewed with a focus on their complexity and the comprehensible presentation of the system.

New Possibilities

The intention of the new approach is to simplify the modelling of event-based communication which should also improve the testing of different system setups. In an additional step, some new exemplary system designs or setups behind the scenarios of the previous case study will be modelled to check if this is true.

8.4. Results

The following paragraphs will discuss the overall result of the evaluation according to the GQM plan presented before.

8.4.1. Q1 Explicitness of the Meta-Model

The existing PCM meta-model did not provide any elements specific to events and event-based communication. As shown in the previous case study [RK09a] a workaround was possible to setup performance equivalent structures. In the presented approach of this thesis, there are meta-model elements for EventTypes, EventGroups, Sinks, Sources, event connectors and event emitting actions for the processing flow. While there is no numerical quality index for such a difference, it can be clearly stated that there is an improvement on the coverage of event related elements, at least in the complexity of the new element modelling and the previous workaround.
The answer to this question based on the second metric is more conclusive. As an example, Figure 8.3 presents a source to sink connection from the first case study regarding the TIME system. Without any additional information, this can not be identified as an event-based or as an asynchronous connection. A closer investigation to the SEFF of the source wrapper reveals the asynchronous ForkAction but this fact will not be known to the architect without first completing this research. Figure 8.4 presents the same event-based connection between the SBUS and the StorageLocation component in the new case study. Now the event-based connection is explicit to the architect and the model is far more straightforward and clear compared to the previous model.

![Figure 8.3: Source Sink Connection - Old Case Study](image1)

![Figure 8.4: Source Sink Connection - New Case Study](image2)

The same applies to the other models improved in the new approach. In the SEFF of the event emitting component, the architect does now explicitly find an EmitEventAction instead of the general operation ExternalCallAction used in the old case study. Furthermore, the repository now includes an explicit EventGroup and SourceRole and SinkRoles instead of the general interface and role elements which did not have any indication of an asynchronous or event-based connection.

The metrics clearly prove that the new approach improves the conclusiveness of the Palladio models for event-based communication.

### 8.4.2. Q2 Reduced Modelling Effort

**Number of Components per Connection**

The number of required model elements to be created to complete a connection between a source an a sink component was analysed. No time assumptions to create the different elements were made for the individual experience and training in the usage of Eclipse modelling tools in general and the PCM tools in particular. The modelling of resource demands is not reflected in this metric because of the independence from the contribution of this approach.

The source and sink components themselves were present and deployed with the according assembly and allocation context elements. The sink component contained the necessary ProvidedRole for the connection. The source component did not contain any required role elements for the connection and no call action in its SEFF.
In the old case study, the SBUS infrastructure required four additional components between the sink and source components (Figure 8.3). Two of the components represented the SBUS wrapper for the source and the sink, the others represented the thread pools necessary to model the single threaded wrappers. While it was necessary to model the source and sink wrappers as individual components, the ThreadPool component was modelled only once in the repository but deployed once per wrapper in the system. The effort tracking led to three new basic components but four new AssemblyContext elements and four new AllocationContext elements being created.

An interface to describe the new connection with a signature to emit an event was added to the repository. RequiredRoles for this interface were added to the source, the source wrapper and the sink wrapper. Additionally the wrappers received a provided role for the interface. Another Interface was added to describe the connection between the wrappers and the thread pools. This interface included two signatures to acquire or release a thread. In addition, required roles for this interface were added to the wrapper and a provided role was added to the thread pool. The SEFF of the source component was extended with an ExternalCall to the new required role. A complete new SEFF with two external calls to the signatures of the thread pool interface was added to each wrapper. A processing SEFF was also added to the thread pool component. The SEFF of the source wrapper included a ForkAction with a ForkedBehaviour element that encapsulated the ExternalCallAction to the sink wrapper. For each ExternalCallAction a VariableUsage element was created to either configure or forward the characteristics of the triggered event. In the TIME scenario the VALUE, TYPE, and NUMBER_OF_ELEMENTS characteristics were set for the events, and according elements were created for every VariableUsage. In order to connect the deployed components, five connectors in total were necessary to link all required roles with the provided roles.

It was necessary to create an EventGroup and an EventType for the new approach to describe the event send between the components. A SourceRole that references this interface and a corresponding EmitEventAction that references to its SEFF were added to the source component. As in the previous case study, a VariableUsage with three VariableCharacteristics elements was added to this EmitEventAction. Finally, an AssemblyEventConnector was added to link the new SourceRole with the existing SinkRole of the sink component.

Table 8.2 compares the tracked effort for the SBUS case studies of the old and the new approaches. In total 59 required new elements for a source to sink connection were used in the old approach, compared to 11 new elements for a connection based on the new approach. This is a clear indicator for the improvement on modelling an event-based communication between components. Even without the specific effort per event creation, the result of the metric highlights the improvement of the new approach.

**Number of Steps for an Additional Sink**

The second metric of question two requires a system with an existing event-based connection between two components as an initial setup. For both approaches the required steps to connect an additional sink component to the present source has been tracked and compared to each other.

With the old approach, it was necessary, to add an additional RequiredRole to the source wrapper component as well as a new ForkedBehaviour with an ExternalCallAction to the SEFF of the source wrapper. The ExternalCallAction was linked with the new RequiredRole and contained a VariableUsage and three VariableCharacterisations as described for the first metric. The Interface element for the RequiredRole was the same as for the existing connection. The additional sink component had a ProvidedRole for this interface and an instance of the component was already deployed in the system. Nevertheless,
new instances of the existing SinkWrapper and the ThreadPool components were necessary and two new AssemblyContexts and AllocationContexts were required for them. Furthermore, three AssemblyConnectors were necessary to connect the new component instances.

With the new approach, and an additional deployed sink component that included a Sink-Role for the according EventGroup, only one new AssemblyEventConnector between the existing SourceRole and the SinkRole was necessary.

Table 8.3 summarises the described steps. Due to the multi-connection capability of the new SourceRole element and the transparency of the connection details, only one step was necessary with the new approach instead of fourteen steps with the old approach.

<table>
<thead>
<tr>
<th>Element</th>
<th>Old Approach</th>
<th>New Approach</th>
</tr>
</thead>
<tbody>
<tr>
<td>Interface / EventGroup</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>Signature / EventType</td>
<td>3</td>
<td>1</td>
</tr>
<tr>
<td>Basic Component</td>
<td>3</td>
<td>0</td>
</tr>
<tr>
<td>Required Role</td>
<td>5</td>
<td>1</td>
</tr>
<tr>
<td>Provided Role</td>
<td>3</td>
<td>0</td>
</tr>
<tr>
<td>SEFF</td>
<td>3</td>
<td>0</td>
</tr>
<tr>
<td>Fork Action</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>Fork Behaviour</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>Call Action</td>
<td>5</td>
<td>1</td>
</tr>
<tr>
<td>VariableUsage</td>
<td>5</td>
<td>3</td>
</tr>
<tr>
<td>VariableCharacterisation</td>
<td>15</td>
<td>3</td>
</tr>
<tr>
<td>Assembly Context</td>
<td>4</td>
<td>0</td>
</tr>
<tr>
<td>Connector</td>
<td>5</td>
<td>1</td>
</tr>
<tr>
<td>Allocation Context</td>
<td>4</td>
<td>0</td>
</tr>
<tr>
<td>Total</td>
<td>59</td>
<td>11</td>
</tr>
</tbody>
</table>

Table 8.3.: Element Creation for an Additional Sink
8.4. Results

Number of Steps for an Additional Source

The third metric to answer question two concerns the required steps to connect a new source component to an existing connection. Once again, a complete connection between a sink and a source component was assumed. A new source component was assembled in the system model and an according allocation context was put in place.

With the old approach, it was necessary to create a new required role on the source component according to the interface of the first connection. In the components SEFF an ExternalCallAction for that role was created with one VariableUsage and three VariableCharacterisation elements. Besides the additional sink component, a new source component also implies a new source wrapper component in the repository. This is because each source component can be connected with a different number of sinks which requires source wrappers with different numbers of ForkedBehaviours in their SEFFs. As a result a new SourceWrapper component was created with two required roles, one for the event emitting interface and one for the ThreadPool interface. The internal SEFF included the ForkAction with the ForkedBehaviour as well as three ExternalCallActions with one VariableUsage and three VariableCharacterisations for each of them. One of these actions calls the required role for the event transmission interface, the others call the acquire and release thread signatures of thread pool interface. Furthermore, a provided role was added to the new source wrapper to satisfy the required role of the source component. The deployment of the new wrapper and the according thread pool required two assembly contexts and two allocation contexts. To connect them to each other, to the source and to the appropriate sink wrapper, three new assembly connectors were added to the system.

With the new approach, a source role was added to the new source component. The SEFF of the source was extended with an EmitEventAction, the according variable usage and three variable characterisations. A new AssemblyEventConnector was added between this source role and the already present sink role for the connection itself.

Table 8.4 summarises the newly created elements. The reduction from 35 new elements previously down to only 6 elements to add an additional source, confirm the modelling improvements provided by the presented approach.

<table>
<thead>
<tr>
<th>Element</th>
<th>Old Approach</th>
<th>New Approach</th>
</tr>
</thead>
<tbody>
<tr>
<td>Interface / EventGroup</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Signature / EventType</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Basic Component</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>Required Role</td>
<td>3</td>
<td>0</td>
</tr>
<tr>
<td>Provided Role</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>SEFF</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>Fork Action</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>Fork Behaviour</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>Call Action</td>
<td>4</td>
<td>1</td>
</tr>
<tr>
<td>VariableUsage</td>
<td>4</td>
<td>1</td>
</tr>
<tr>
<td>VariableCharacterisation</td>
<td>12</td>
<td>3</td>
</tr>
<tr>
<td>Assembly Context</td>
<td>2</td>
<td>0</td>
</tr>
<tr>
<td>Connector</td>
<td>3</td>
<td>1</td>
</tr>
<tr>
<td>Allocation Context</td>
<td>2</td>
<td>0</td>
</tr>
<tr>
<td><strong>Total</strong></td>
<td><strong>35</strong></td>
<td><strong>6</strong></td>
</tr>
</tbody>
</table>

Table 8.4.: Element Creation for an Additional Source

Figure 8.5 provides a combined overview of the results of all three metrics. The bars of the
8. Evaluation

The chart represents the number of new elements to be created for each approach per metric. The diagram visualises the reduced effort needed in all three cases with the new approach.

![Chart showing the number of elements created per approach.](image)

**Figure 8.5.: Metric Summary: Number of Elements to Create per Analysed Case**

### 8.4.3. Q3 Validity of the Simulation Results

Besides the improved modelling capabilities when compared to the old approach, it is also necessary to provide a quality prediction that at least meets the previous standards. To validate this statement, the scenarios of the previous case study have been analysed again and compared with the new approach. The new prediction results were compared to those of the old case study as well as to the original measurements that were part of the first case study.

The measurements were performed in a laboratory of the Karlsruhe Institute of Technology (KIT) with a server cluster of identical machines, each equipped with a 2.4 GHz Intel Core2Quad Q6600 CPU, 8 GB main memory, and two 500 GB SATA II hard discs. All machines were running Ubuntu Linux version 8.04 and were connected to a GBit LAN.

Four different deployments with singular and redundant installed components were analysed. To generate a load on the system, the recorded data from the city of Cambridge was replayed to trigger the services provided by the ACIS and SCOOT components. The CPU utilisation was predicted and compared to the measurements for all scenarios. Additionally, the mean processing time of the location storage was analysed for scenario 2 to gain more insights about the high utilisation on server 1.

To estimate the resource demands of the TIME and the middleware components, they were extended with internal sensors and installed in the laboratory at the KIT. After the installation, their mean processing time for a set of events was measured under a very low load to obtain their individual processing times. Table 8.5 presents the measured resource demands. To model these demands in the new case study, the measured times were used as CPU resource demands of the according internal actions in the repository components. The SBUS middleware components were measured once for each TIME component and the average processing time was used for their resource demand in the Palladio model.

The processing time spent, identified as “time in component” in Table 8.5, was modelled as CPU resource demands for an InternalAction in the SEFF of the TIME specific components. The same was repeated for the processing times to create the client request of...
8.4. Results

the ProximityDetection and to create the server response in the SCOOT component. The other values were configured in the SBUS middleware repository model that was weaved into the system during the simulation process. The mean of the source “time in library” values was placed as a CPU resource demand of an InternalAction in the SEFF that handles the signature of the IMiddlewareSourcePort interface. The mean of the source “time in wrapper” was configured as a CPU resource demand in the SEFF for the IMiddlewareEventDistribution interface. The SourceWrapper component in the middleware repository provided both of these interfaces.

The sink “time in wrapper” was configured as a CPU resource demand in the IMiddlewareEventReceiver handling the SEFF of the SinkWrapper component in the SBUS middleware repository. In the same way, the “time in library” was configured in the SEFF that handles the IMiddlewareSinkPort interface.

The single threaded characteristics of the TIME and SBUS components were modelled with passive resources in the appropriate repository components.

<table>
<thead>
<tr>
<th>Component</th>
<th>Endpoint</th>
<th>Time in Component</th>
<th>Source</th>
<th>Sink</th>
<th>Client Request</th>
<th>Server Response</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>Time in Library</td>
<td>Time in Wrapper</td>
<td>Time in Library</td>
<td>Time in Wrapper</td>
<td></td>
</tr>
<tr>
<td>ACIS</td>
<td>feeds</td>
<td>0.5172</td>
<td>0.0369</td>
<td>0.0097</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Location</td>
<td>feeds</td>
<td>0.6343</td>
<td></td>
<td>0.0088</td>
<td>0.0088</td>
<td></td>
</tr>
<tr>
<td>SCOOT</td>
<td>lightred</td>
<td>0.6266</td>
<td>0.04</td>
<td>0.0167</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>lightgreen</td>
<td>0.6266</td>
<td>0.04</td>
<td>0.0192</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>linkinfo</td>
<td>0.5225</td>
<td></td>
<td></td>
<td>0.03170</td>
<td></td>
</tr>
<tr>
<td>Proximity</td>
<td>lightred</td>
<td>0.4511</td>
<td></td>
<td>0.0005</td>
<td>0.0072</td>
<td></td>
</tr>
<tr>
<td></td>
<td>lightgreen</td>
<td>0.3139</td>
<td></td>
<td>0.0005</td>
<td>0.0072</td>
<td></td>
</tr>
<tr>
<td></td>
<td>linkinfo</td>
<td>0</td>
<td></td>
<td></td>
<td>0.02870</td>
<td></td>
</tr>
</tbody>
</table>

Table 8.5.: Measured Resource Demands of the Components (in ms)

The comparison of the prediction and measurement results for all scenarios will be presented in the following paragraphs. The new approach slightly improved on the prediction results of the old approach. Only in scenario 1 and for server 1 in scenario 2 did the old approach give predictions that were 0.1 to 2.8 percentage points better. All of the results are still very good and the prediction error never exceeded 8.3 percentage compared to the actual measurements.

8.4.3.1. Scenario 1

In scenario 1 a single instance of each TIME component was deployed. The ACIS and LocationStorage components were installed on server 1. SCOOT and ProximityDetection were installed on server 2. ACIS and SCOOT were then triggered with different load levels. Figure 8.6 shows the graphs of the original measurements, the prediction of the old case study and the prediction with the new approach. The measurement and prediction data is provided in appendix A. The data of scenario 1 can be found in table A.1.

The average difference from the new prediction to the original measurements is 0.9 percentage points, which is half of the average difference between the old prediction and respective measurements. The maximum difference in the old case study was about 4.7 percentage
points. The maximum difference in the new case study was only 1.8 percentage points. For server 2, the old case study provided an average prediction error of 1.2 percentage points and a maximum error of 3.5 percentage points. In the new case study the average error was 3.1 percentage points and the maximum error 4.9 percentage points. In summary for scenario 1, the prediction result was of the same quality as in the first case study.

![Scenario 1 - CPU Utilisation Server 1](image1)

![Scenario 1 - CPU Utilisation Server 2](image2)

Figure 8.6.: Scenario 1 - CPU Utilisation

8.4.3.2. Scenario 2

Scenario 2 was designed to produce a higher load on the hardware. To achieve this three instances of each component were installed instead of one single instance. Three ACIS and LocationStorages on server 1 and three SCOOT and ProximityDetection instances on server 2 were installed. The load on ACIS and SCOOT were also tripled by triggering each of them with the same load. The applied load levels and the results for server 1 are presented in Table A.2 and Table A.3 for server 2. The event rate documented in these tables are per SCOOT and per ACIS respectively.

Figure 8.7 presents the CPU utilisations of server 1 and server 2 for the individual load levels. In the first case study, the average prediction error for server 1 was 0.8 percentage points and the maximum prediction error was 1.6 percentage points. With the new approach, the maximum prediction error for the server was 8.3 and the average prediction error was 3.1 percentage points.
error was 3.6 percentage points. For server 2 the difference between the old and the new prediction quality was even smaller. With the old approach, it was about 0.9 percentage points and with the new one, 1.0 percentage points. The maximum error recorded was 1.2 percentage points with the old and 1.5 percentage points with the new approach.

![Graph](image)

**Figure 8.7.: Scenario 2 - CPU Utilisation**

Scenario 2 generates a high load on the system. To gain additional insights regarding the higher loads, the mean processing time of the location storage component was also analysed. Figure [8.8] presents the results of the measurement as well as the prediction for the old and the new case study. Table [A.2] provides the detailed measurement and prediction data. For the mean processing time of this scenario, the prediction with the new approach was even better compared to the old approach. The average prediction error was 4.7 percentage points instead of 5.5 and the maximum prediction error was 6.9 instead of 8.0 percentage points in the old case study.

### 8.4.3.3. Scenario 3

The purpose of scenario 3 was to analyse the contention effects with an instance of all components installed on a single server. ACIS and SCOOT were triggered in parallel with seven different load levels. The measurement and prediction data can be found in Table [A.4] in the appendix. Figure [8.9] presents the results of the measurement and the predictions.
Once again the new approach has a smaller prediction error than the old approach. While the average prediction error in the old case study was 1.8 and the maximum error 3.4 percentage points, the new approach predicted the CPU utilisation with an average error of only 1.3 percentage points and a maximum error of only 2.6 percentage points.

The last scenario was designed to analyse the contention effects of the third scenario with a higher load. To accomplish two instances of ACIS, LocationStorage, SCOOT and ProximityDetection were installed on a single server. As shown in Figure 8.10, the prediction results were slightly better with the new approach. The average prediction error was 2.6 compared to 4.0 percentage points in the old case study. The same applies to the maximum error which was only 5.8 instead of 8.5 percentage points in the old case study.

8.4.4. Q4 Testing Different Middleware Setups/Settings

In addition to the improvement of modelling and predictions of existing solutions, the process should be simplified to model new configurations and architectures to predict changes before their implementation. To answer this question, the same metrics from question two were used. They provided insights on the complexity to build a new event-based communication or to change an existing one. On the other hand, metrics which
provide information about the complexity to change a middleware configuration or to set it up are utilised.

The metrics regarding the changing of the system itself have indicated a clear advantage by using the new approach presented in this thesis. Figure 8.5 provides a overview of the reduced effort for the different tasks of the metrics.

Two different scenarios were investigated to analyse the effort required to change the middleware. In the first case, the configuration of the existing middleware was changed. In the second case, the middleware was replaced completely. Scenario 1 of the TIME case study was assumed to estimate the effort required to change the middleware.

8.4.4.1. Middleware Adaptation

The first adaptation is the change of the middleware resource demands. The resource demands of the SBUS sink and source wrapper need to be adopted. As presented in Table 8.5, the resource demands are spread over multiple InternalActions. In the old approach these internal actions were part of the common repository. In the new approach they are encapsulated in the middleware repository. In the specific scenario that was analysed, the required effort for the modification was exactly the same. Both repositories contained only one unique instance for the source, the sink wrapper components.

Nevertheless, in a scenario with multiple source wrappers connected to a different number of sink components, the old approach would require individually modelled source wrapper components for every source. In such a case, the adaptation of the source wrapper would have been required for every source. In contrast to this, the new approach requires only the same singular modification, independent from the number and degree of sink to source connections.

The second adaptation is the change of the SBUS middleware to a multi-threaded system. With the new approach, only the SourceWrapper and the SinkWrapper components in the middleware repository needed to be adopted. This implies the deletion of the components passive resources and the corresponding four acquire and release action elements. There are six modifications in total without the changes to the control flow references in the respective SEFFs.

With the old approach it would have been necessary to remove the ThreadPool components, assemblies and allocations as well as the corresponding required roles and external call actions from the wrapper components. Omitting the required reconfiguration of data flow references in the SEFFs led to 15 modifications as presented in Table 8.6.
While the improvement for the first adaptation applied only in specific scenarios, the modification is clearly better in all scenarios supported with the approach presented in this thesis. Especially for larger systems with more connections, varying source to sink relationships and middleware components which are not already deployed, the required effort with the old approach raised dramatically.

With the new approach, the effort is constant and independent from the number and complexity of source to sink connections. There is also no manual effort needed to deploy the middleware components.

<table>
<thead>
<tr>
<th>Modification</th>
<th>Old Approach</th>
<th>New Approach</th>
</tr>
</thead>
<tbody>
<tr>
<td>Delete Passive Resource</td>
<td>0</td>
<td>2</td>
</tr>
<tr>
<td>Delete Component</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>Delete Assembly Context</td>
<td>4</td>
<td>0</td>
</tr>
<tr>
<td>Delete Allocation Context</td>
<td>4</td>
<td>0</td>
</tr>
<tr>
<td>Delete ExternalCallAction</td>
<td>4</td>
<td>0</td>
</tr>
<tr>
<td>Delete RequiredRole</td>
<td>2</td>
<td>0</td>
</tr>
<tr>
<td>Delete Acquire / Release Action</td>
<td>0</td>
<td>4</td>
</tr>
<tr>
<td><strong>Total</strong></td>
<td><strong>15</strong></td>
<td><strong>6</strong></td>
</tr>
</tbody>
</table>

Table 8.6.: Effort Removing Single Threaded Characteristic

### 8.4.4.2. Middleware Replacement

The effort required to replace the middleware with another product depends on the specific middleware to be used to replace the old one. Only the effort to connect the source and sink components to the middleware is considered and not the effort to model the middleware itself. Furthermore, we assume that the new middleware interfaces are compatible with the existing ones.

Even with these strong assumptions that ignore the effort to create and deploy the middleware components, the required effort is considerably higher with the old approach than with the new one. Table 8.7 summarises the required effort. With the old approach it was necessary to remove the allocation and assembly contexts for the old SBUS wrapper and thread pool components. Afterwards, the components needed to be removed from the repository as well. To integrate the new middleware, at least 4 AssemblyConnectors were necessary to connect the required and provided roles of the TIME components. As mentioned before, this effort represents the minimum required effort. At least the assembly and allocation contexts for the middleware components would be necessary in addition this.

With the new approach, it is only necessary to select another middleware repository to be automatically weaved into the model during the prediction process.

<table>
<thead>
<tr>
<th>Modification</th>
<th>Old Approach</th>
<th>New Approach</th>
</tr>
</thead>
<tbody>
<tr>
<td>Delete Assembly Context</td>
<td>8</td>
<td>0</td>
</tr>
<tr>
<td>Delete Allocation Context</td>
<td>8</td>
<td>0</td>
</tr>
<tr>
<td>Delete Component</td>
<td>3</td>
<td>0</td>
</tr>
<tr>
<td>Add Assembly Connector</td>
<td>4</td>
<td>0</td>
</tr>
<tr>
<td>Configure Other Middleware</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td><strong>Total</strong></td>
<td><strong>23</strong></td>
<td><strong>1</strong></td>
</tr>
</tbody>
</table>

Table 8.7.: Replace Middleware
The overall answer to question 4 based on the related metrics is visualised in Figure 8.11. For all of these metrics, there is only one scenario for which the metrics produces at least the same results for the old and the new approaches. In all the other cases, the new approach improves the prediction of new setups and configurations of systems which make use of an event-based communication.

![Exemplary Modelling Efforts to Test New Setups](image)
9. Conclusion

9.1. Summary

In this thesis, a novel approach to model and predict event-based communication in component-based software architectures has been presented. A meta-model extension for the Palladio Component Model with advanced editing capabilities has been combined with a mapping and model-to-model transformation to enable a quality prediction based on existing simulation capabilities. The approach has been implemented as a ready-to-use infrastructure integrated in the Palladio workbench. The implementation has been evaluated in comparison to an existing reference case study.

In the following the main contribution of the thesis are summarised.

Meta-Model

The PCM meta-model has been refactored to allow different types of component connections to explicitly model event-based communication in component based software architectures. According to this new infrastructure, elements to model event-based communication have been introduced in the meta-model. Furthermore, the graphical editing capabilities have been extended to support the new model elements. As shown in the evaluation, the new approach provides improved modelling capabilities to the software architect with less effort and more explicit semantics.

Transformation and Middleware Weaving

To support quality predictions for the new meta-model elements, a mapping from the new elements to a set of elements already supported by the simulation facilities has been defined. According to this mapping, a model-to-model transformation was developed to automatically transform an extended PCM model to a classic one. Furthermore, to test and predict the influence of different middleware products and configurations, the transformation has been designed to weave in an additional middleware model. This model encapsulates the characteristics of the middleware used to implement the event-based communication.

The evaluation has shown, that the separation of the middleware and the system model provides enhanced modelling capabilities as well as high quality simulation results with deviations considerably lower then 8% for the majority of test cases.
9. Conclusion

Integration in the PCM

The developed meta-model, as well as the transformation, have been fully integrated in the Palladio Workbench. Additional enhancements for the PCM infrastructure such as the Completion Framework have therefore been developed. Furthermore, all parts of the integration can be used separately. For example, the new modelling capabilities can be used independently from the transformation. The integration was extensively used and tested during the evaluation of this thesis.

9.2. Open Issues

The presented approach provides a self-contained solution. Nevertheless, there are some open issues that were not necessary for the presented case study but represent good enhancements for the future.

9.2.1. Source Delegation Connector

The extended PCM meta-model and the generated data-model already provide event delegation connectors to connect external and internal SourceRoles and SinkRoles in the case of ComposedStructures. So far, these connectors and the external Source- and SinkRoles are neither supported by the graphical editors nor by the transformation.

Nevertheless, a user would be able to use the automatically generated tree editors to add delegation connectors to a PCM model instance. In this case, the connector can not be presented in a graphical representation of the model and the simulation process fails because of the unknown elements.

9.2.2. Complex Event Forwarding

It is possible to model complex event types using a complex data type for the event content. This could also be accessed in the source and in the sink components and would be present in the OperationInterfaces which replace the original EventGroups. As a limitation, the presented transformation does not include an analysis of complex parameters passed to one of the intermediate components and only the characteristics of the root parameter are forwarded. In addition, the parameters passed to the signatures of the middleware interfaces are fixed primitive data types and are therefore not able to transport the structure of a complex data type.

The presented approach makes the assumption that sinks do not rely on characteristics of the compartments in case of a complex data type. If this is the case, the simulation might fail because of the missing characterisation.

9.2.3. Post Processing Middleware Hook

In the presented approach, the asynchronous communication has been split into two decoupled synchronous operation calls. One operation call from the source to the event distribution and one call from the event distribution to the sink. The latter may exist once per sink. In the intermediate model, every involved component performs an external call to the appropriate middleware component before the call is forwarded to the downstream communication component. It is assumed, that the middleware call before the call forwarding is sufficient and that there therefore there is no post processing call as a hook into the middleware. If a system is targeted in the future which requires such a hook, then it would require just a small effort to add such a hook. It was omitted from the thesis to not inflate the model while it is unclear if such a hook will ever be necessary.
9.3. Future Development

The development of the presented approach has raised some topics which are candidates for further investigations. A brief overview of these topics is given below.

9.3.1. Filter Distribution

To filter events which should not be processed by a sink a `BranchAction` can be used in the event handler. Nevertheless, some platforms allow one to publish such filters to the event distribution. If a filter is already applied during the distribution it might prevent unnecessary network traffic and resource demands.

It would be possible to analyse the RDSEFFs of the event handlers within the transformation process and to publish a possible event filtering `BranchAction` to the EventSender or EventDistribution components.

9.3.2. Simulation Result Transformation

In the presented approach, the original model is transformed into a more fine-grained and complex model of the analysed software. Accordingly to this, the simulation results contain a considerable overhead of information because of the intermediate components, which might not be handy for the user.

To support the user in this matter, there should be a reverse transformation of the simulation results to match the original high-level architecture.

9.3.3. Probes

To limit the complexity of the simulation results in association with the reverse transformation of the simulation results, a new framework in the PCM to specify probes for the analysed model has been developed. These probes can be used to specify measurement points in the model. This can also reduce the number of unnecessary measurement points in the generated component chain. This would also accelerate the simulation run because less information has to be processed.

9.3.4. SOA and SLAs

In the area of Service Oriented Architectures (SOA) event-based communication and the according quality predictions become more and more relevant to predict the effects of the complex structures and the inter-service impacts. Not only the necessity to check the feasibility of Service Level Agreements (SLA) requires solutions such as the one presented in this approach. Some research should be investigated to discover additional requirements and opportunities to predict these architectures.
A. Case Study Data

The following tables include the simulation results and measurements from the old and the new case studies.
### A.1. Scenario 1

Scenario with one ACIS and one LocationStorage instances on server 1 and one SCOOT and one ProximityDetection on server 2.

<table>
<thead>
<tr>
<th>Load Level</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>SERVER 1</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ACIS</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Events/Sec</td>
<td>86,82</td>
<td>153,36</td>
<td>399,88</td>
<td>787,32</td>
<td>1197,6</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Inter-arrival Time</td>
<td>86,82</td>
<td>153,36</td>
<td>399,88</td>
<td>787,32</td>
<td>1197,6</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Measurement</td>
<td>3,55%</td>
<td>6,40%</td>
<td>12,70%</td>
<td>24,75%</td>
<td>36,20%</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Prediction Old</td>
<td>3,00%</td>
<td>5,30%</td>
<td>12,10%</td>
<td>26,90%</td>
<td>40,90%</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Busy 1 Job</td>
<td>10,50%</td>
<td>18,50%</td>
<td>48,20%</td>
<td>94,90%</td>
<td>56,50%</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Busy 2 Jobs</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>43,50%</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Prediction New</td>
<td>2,63%</td>
<td>4,63%</td>
<td>12,05%</td>
<td>23,73%</td>
<td>35,88%</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Diff. New to Classic</td>
<td>0,4%</td>
<td>0,7%</td>
<td>0,1%</td>
<td>3,2%</td>
<td>5,0%</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Diff. New to Measurement</td>
<td>0,9%</td>
<td>1,8%</td>
<td>0,7%</td>
<td>1,0%</td>
<td>0,3%</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Diff. Classic to Measurement</td>
<td>0,6%</td>
<td>1,1%</td>
<td>0,6%</td>
<td>2,2%</td>
<td>4,7%</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>SERVER 2</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>SCOOT</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Events / Sec</td>
<td>95,05</td>
<td>182,02</td>
<td>391,45</td>
<td>660,18</td>
<td>976,4</td>
<td>1809,9</td>
<td>1880,2</td>
<td>1917,85</td>
</tr>
<tr>
<td>Inter-arrival Time</td>
<td>0,01052</td>
<td>0,00549</td>
<td>0,00255</td>
<td>0,00151</td>
<td>0,00102</td>
<td>0,00055</td>
<td>0,00053</td>
<td>0,00052</td>
</tr>
<tr>
<td>Measurement</td>
<td>3,40%</td>
<td>6,40%</td>
<td>11,60%</td>
<td>21,20%</td>
<td>29,00%</td>
<td>41,40%</td>
<td>45,20%</td>
<td>44,40%</td>
</tr>
<tr>
<td>Prediction Classic</td>
<td>2,90%</td>
<td>5,50%</td>
<td>11,80%</td>
<td>19,40%</td>
<td>28,60%</td>
<td>39,80%</td>
<td>45,80%</td>
<td>47,90%</td>
</tr>
<tr>
<td>Busy 1 Job</td>
<td>12,90%</td>
<td>24,70%</td>
<td>53,00%</td>
<td>76,00%</td>
<td>66,30%</td>
<td>2,20%</td>
<td>1,60%</td>
<td>2,80%</td>
</tr>
<tr>
<td>Busy 2 Jobs</td>
<td>6,70%</td>
<td>32,90%</td>
<td>97,80%</td>
<td>98,40%</td>
<td>97,20%</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Prediction New</td>
<td>3,23%</td>
<td>6,18%</td>
<td>13,25%</td>
<td>22,35%</td>
<td>33,03%</td>
<td>49,45%</td>
<td>49,60%</td>
<td>49,30%</td>
</tr>
<tr>
<td>Diff. New to Classic</td>
<td>0,3%</td>
<td>0,7%</td>
<td>1,5%</td>
<td>3,0%</td>
<td>4,4%</td>
<td>9,7%</td>
<td>3,8%</td>
<td>1,4%</td>
</tr>
<tr>
<td>Diff. New to Measurement</td>
<td>0,2%</td>
<td>0,2%</td>
<td>1,7%</td>
<td>1,2%</td>
<td>4,0%</td>
<td>8,1%</td>
<td>4,4%</td>
<td>4,9%</td>
</tr>
<tr>
<td>Diff. Classic to Measurement</td>
<td>0,5%</td>
<td>0,9%</td>
<td>0,2%</td>
<td>1,8%</td>
<td>0,4%</td>
<td>1,6%</td>
<td>0,6%</td>
<td>3,5%</td>
</tr>
</tbody>
</table>

Table A.1.: Scenario 1 Results
## A.2. Scenario 2

Scenario with three ACIS and three LocationStorage instances on server 1 and three SCOOT as well as three ProximityDetection on server 2.

<table>
<thead>
<tr>
<th>Load Level</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>SERVER 1</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>ACIS</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Events/Sec</td>
<td>95,05</td>
<td>182,02</td>
<td>391,45</td>
<td>660,18</td>
<td>976,4</td>
<td>1809,9</td>
</tr>
<tr>
<td>Inter-arrival Time</td>
<td>0,01052</td>
<td>0,00549</td>
<td>0,00255</td>
<td>0,00151</td>
<td>0,00102</td>
<td>0,00055</td>
</tr>
<tr>
<td><strong>CPU Util.</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Measurement</td>
<td>9,70%</td>
<td>18,00%</td>
<td>35,60%</td>
<td>57,60%</td>
<td>80,59%</td>
<td>91,59%</td>
</tr>
<tr>
<td>Prediction Old</td>
<td>8,60%</td>
<td>16,40%</td>
<td>35,60%</td>
<td>58,30%</td>
<td>80,30%</td>
<td>92,50%</td>
</tr>
<tr>
<td>Busy 3 Job</td>
<td>11,50%</td>
<td>22,00%</td>
<td>47,30%</td>
<td>79,80%</td>
<td>45,50%</td>
<td>0,30%</td>
</tr>
<tr>
<td>Busy 6 Jobs</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>54,50%</td>
<td>97,70%</td>
</tr>
<tr>
<td>Busy 9 Jobs</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>2,00%</td>
</tr>
<tr>
<td><strong>Prediction New</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Measurement</td>
<td>8,63%</td>
<td>16,50%</td>
<td>35,48%</td>
<td>59,85%</td>
<td>88,63%</td>
<td>99,93%</td>
</tr>
<tr>
<td>Diff. New to Classic</td>
<td>0,0%</td>
<td>0,1%</td>
<td>0,1%</td>
<td>1,6%</td>
<td>8,3%</td>
<td>7,4%</td>
</tr>
<tr>
<td>Diff. New to Measurement</td>
<td>1,1%</td>
<td>1,5%</td>
<td>0,1%</td>
<td>2,3%</td>
<td>8,0%</td>
<td>8,3%</td>
</tr>
<tr>
<td>Diff. Classic to Measurement</td>
<td>1,1%</td>
<td>1,6%</td>
<td>0,0%</td>
<td>0,7%</td>
<td>0,3%</td>
<td>0,9%</td>
</tr>
<tr>
<td><strong>MPT</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Measurement</td>
<td>0,657</td>
<td>0,695</td>
<td>0,676</td>
<td>0,703</td>
<td>0,780</td>
<td>0,846</td>
</tr>
<tr>
<td>Prediction Old</td>
<td>0,634</td>
<td>0,634</td>
<td>0,634</td>
<td>0,634</td>
<td>0,700</td>
<td>0,855</td>
</tr>
<tr>
<td>Instance 1</td>
<td>0,634</td>
<td>0,634</td>
<td>0,634</td>
<td>0,634</td>
<td>0,820</td>
<td>0,000</td>
</tr>
<tr>
<td>Instance 2</td>
<td>0,634</td>
<td>0,634</td>
<td>0,634</td>
<td>0,634</td>
<td>0,820</td>
<td></td>
</tr>
<tr>
<td>Instance 3</td>
<td>0,634</td>
<td>0,634</td>
<td>0,634</td>
<td>0,634</td>
<td>0,820</td>
<td></td>
</tr>
<tr>
<td><strong>Prediction New</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Measurement</td>
<td>0,634</td>
<td>0,634</td>
<td>0,634</td>
<td>0,634</td>
<td>0,820</td>
<td>0,000</td>
</tr>
<tr>
<td>Diff. New to Classic</td>
<td>0,0%</td>
<td>0,1%</td>
<td>0,1%</td>
<td>1,6%</td>
<td>8,3%</td>
<td>7,4%</td>
</tr>
<tr>
<td>Diff. New to Measurement</td>
<td>1,1%</td>
<td>1,5%</td>
<td>0,1%</td>
<td>2,3%</td>
<td>8,0%</td>
<td>8,3%</td>
</tr>
<tr>
<td>Diff. Classic to Measurement</td>
<td>1,1%</td>
<td>1,6%</td>
<td>0,0%</td>
<td>0,7%</td>
<td>0,3%</td>
<td>0,9%</td>
</tr>
</tbody>
</table>

Table A.2.: Scenario 2 Results Server 1
<table>
<thead>
<tr>
<th>Load Level</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>SERVER 2</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>SCOOT</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Events / Sec</td>
<td>86.96</td>
<td>154</td>
<td>398.6</td>
<td>792</td>
</tr>
<tr>
<td>Inter-arrival Time</td>
<td>0.01150</td>
<td>0.00649</td>
<td>0.00251</td>
<td>0.00126</td>
</tr>
<tr>
<td>CPU Util.</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Measurement</td>
<td>9.94%</td>
<td>16.95%</td>
<td>40.21%</td>
<td>82.37%</td>
</tr>
<tr>
<td>Prediction Classic</td>
<td>8.90%</td>
<td>15.80%</td>
<td>40.80%</td>
<td>81.60%</td>
</tr>
<tr>
<td>Busy 1 Job</td>
<td>2.00%</td>
<td>4.90%</td>
<td>10.90%</td>
<td>5.10%</td>
</tr>
<tr>
<td>Busy 2 Jobs</td>
<td>2.10%</td>
<td>3.40%</td>
<td>12.00%</td>
<td>5.40%</td>
</tr>
<tr>
<td>Busy 3 Jobs</td>
<td>10.00%</td>
<td>17.10%</td>
<td>42.40%</td>
<td>43.90%</td>
</tr>
<tr>
<td>Busy 4 Jobs</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Busy 5 Jobs</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Busy 6 Jobs</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Prediction New</td>
<td>9.05%</td>
<td>15.75%</td>
<td>40.53%</td>
<td>80.90%</td>
</tr>
<tr>
<td>Diff. New to Classic</td>
<td>0.2%</td>
<td>0.1%</td>
<td>0.3%</td>
<td>0.7%</td>
</tr>
<tr>
<td>Diff. New to Measurement</td>
<td>0.9%</td>
<td>1.2%</td>
<td>0.3%</td>
<td>1.5%</td>
</tr>
<tr>
<td>Diff. Classic to Measurement</td>
<td>1.0%</td>
<td>1.2%</td>
<td>0.6%</td>
<td>0.8%</td>
</tr>
</tbody>
</table>

Table A.3.: Scenario 2 Results Server 2
A.3. Scenario 3

Scenario with one instance of ACIS, LocationStorage, SCOOT and ProximityDetection installed on server 1.

<table>
<thead>
<tr>
<th>Load Level</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
</tr>
</thead>
<tbody>
<tr>
<td>SERVER 1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ACIS</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Events/Sec</td>
<td>94.81</td>
<td>180.05</td>
<td>393.25</td>
<td>940.1</td>
<td>1328.39</td>
<td>1144.95</td>
<td>1084.29</td>
</tr>
<tr>
<td>Inter-arrival Time</td>
<td>0.01055</td>
<td>0.00555</td>
<td>0.00254</td>
<td>0.00106</td>
<td>0.00075</td>
<td>0.00087</td>
<td>0.00092</td>
</tr>
<tr>
<td>SCOOT</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Events / Sec</td>
<td>106.8</td>
<td>153.18</td>
<td>396.39</td>
<td>776.68</td>
<td>777.15</td>
<td>943.15</td>
<td>1145.35</td>
</tr>
<tr>
<td>Inter-arrival Time</td>
<td>0.00936</td>
<td>0.00653</td>
<td>0.00252</td>
<td>0.00129</td>
<td>0.00129</td>
<td>0.00106</td>
<td>0.00087</td>
</tr>
<tr>
<td>CPU Util.</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Measurement</td>
<td>6.81%</td>
<td>11.90%</td>
<td>26.50%</td>
<td>53.66%</td>
<td>63.78%</td>
<td>65.14%</td>
<td>70.81%</td>
</tr>
<tr>
<td>Prediction Old</td>
<td>6.50%</td>
<td>10.70%</td>
<td>25.30%</td>
<td>55.10%</td>
<td>66.40%</td>
<td>67.40%</td>
<td>74.20%</td>
</tr>
<tr>
<td>Busy 1 Job</td>
<td>22.50%</td>
<td>33.40%</td>
<td>50.10%</td>
<td>10.00%</td>
<td>4.40%</td>
<td>2.00%</td>
<td></td>
</tr>
<tr>
<td>Busy 2 Jobs</td>
<td>1.70%</td>
<td>4.50%</td>
<td>25.50%</td>
<td>63.80%</td>
<td>35.70%</td>
<td>41.80%</td>
<td>29.70%</td>
</tr>
<tr>
<td>Busy 3 Jobs</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Busy 4 Jobs</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Prediction New</td>
<td>6.48%</td>
<td>10.60%</td>
<td>25.28%</td>
<td>54.60%</td>
<td>66.38%</td>
<td>66.45%</td>
<td>71.98%</td>
</tr>
<tr>
<td>Diff. New to Classic</td>
<td>0.0%</td>
<td>0.1%</td>
<td>0.0%</td>
<td>0.5%</td>
<td>0.0%</td>
<td>1.0%</td>
<td>2.2%</td>
</tr>
<tr>
<td>Diff. New to Measurement</td>
<td>0.3%</td>
<td>1.3%</td>
<td>1.2%</td>
<td>0.9%</td>
<td>2.6%</td>
<td>1.3%</td>
<td>1.2%</td>
</tr>
<tr>
<td>Diff. Classic to Measurement</td>
<td>0.3%</td>
<td>1.2%</td>
<td>1.2%</td>
<td>1.4%</td>
<td>2.6%</td>
<td>2.3%</td>
<td>3.4%</td>
</tr>
</tbody>
</table>

Table A.4.: Scenario 3 Results
Scenario with two instances of ACIS, LocationStorage, SCOOT and ProximityDetection installed on server 1.

<table>
<thead>
<tr>
<th>Load Level</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
</tr>
</thead>
<tbody>
<tr>
<td>SERVER 1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ACIS</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Events/Sec</td>
<td>94.57</td>
<td>178.3</td>
<td>578.5</td>
<td>711.74</td>
<td>764.65</td>
</tr>
<tr>
<td>Inter-arrival Time</td>
<td>0.01057</td>
<td>0.00561</td>
<td>0.00173</td>
<td>0.00141</td>
<td>0.00131</td>
</tr>
<tr>
<td>SCOOT</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Events / Sec</td>
<td>86.76</td>
<td>153.1</td>
<td>394.38</td>
<td>545.19</td>
<td>609.98</td>
</tr>
<tr>
<td>Inter-arrival Time</td>
<td>0.01153</td>
<td>0.00653</td>
<td>0.00254</td>
<td>0.00183</td>
<td>0.00164</td>
</tr>
<tr>
<td>CPU Util.</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Measurement</td>
<td>12.81%</td>
<td>23.04%</td>
<td>61.91%</td>
<td>82.94%</td>
<td>93.55%</td>
</tr>
<tr>
<td>Prediction Old</td>
<td>11.70%</td>
<td>21.40%</td>
<td>65.80%</td>
<td>91.40%</td>
<td>98.50%</td>
</tr>
<tr>
<td>Busy 1 Job</td>
<td>2.70%</td>
<td>4.00%</td>
<td>3.80%</td>
<td>2.90%</td>
<td>1.40%</td>
</tr>
<tr>
<td>Busy 2 Jobs</td>
<td>19.00%</td>
<td>30.80%</td>
<td>41.60%</td>
<td>23.50%</td>
<td>12.40%</td>
</tr>
<tr>
<td>Busy 3 Jobs</td>
<td>0.30%</td>
<td>0.90%</td>
<td>9.70%</td>
<td>16.60%</td>
<td>17.80%</td>
</tr>
<tr>
<td>Busy 4 Jobs</td>
<td>1.20%</td>
<td>3.90%</td>
<td>32.40%</td>
<td>54.70%</td>
<td>62.10%</td>
</tr>
<tr>
<td>Busy 5 Jobs</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Busy 6 Jobs</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Prediction New</td>
<td>11.60%</td>
<td>20.98%</td>
<td>61.43%</td>
<td>79.63%</td>
<td>87.70%</td>
</tr>
<tr>
<td>Diff. New to Classic</td>
<td>0.1%</td>
<td>0.4%</td>
<td>4.4%</td>
<td>11.8%</td>
<td>10.8%</td>
</tr>
<tr>
<td>Diff. New to Measurement</td>
<td>1.2%</td>
<td>2.1%</td>
<td>0.5%</td>
<td>3.3%</td>
<td>5.8%</td>
</tr>
<tr>
<td>Diff. Classic to Measurement</td>
<td>1.1%</td>
<td>1.6%</td>
<td>3.9%</td>
<td>8.5%</td>
<td>5.0%</td>
</tr>
</tbody>
</table>

Table A.5.: Scenario 4 Results
## List of Figures

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.1</td>
<td>OMG Performance Prediction Process [Bec08b]</td>
<td>10</td>
</tr>
<tr>
<td>2.2</td>
<td>Performance Impacts</td>
<td>12</td>
</tr>
<tr>
<td>2.3</td>
<td>Palladio Meta-Model and DSLs</td>
<td>13</td>
</tr>
<tr>
<td>2.4</td>
<td>Providing Requiring Entity</td>
<td>14</td>
</tr>
<tr>
<td>2.5</td>
<td>Interfaces and Signatures</td>
<td>14</td>
</tr>
<tr>
<td>2.6</td>
<td>Composed Structure</td>
<td>16</td>
</tr>
<tr>
<td>2.7</td>
<td>Usage Model</td>
<td>17</td>
</tr>
<tr>
<td>2.8</td>
<td>User Model Actions</td>
<td>17</td>
</tr>
<tr>
<td>2.9</td>
<td>SimuCom Workflow</td>
<td>19</td>
</tr>
<tr>
<td>2.10</td>
<td>MOF Layer Example</td>
<td>20</td>
</tr>
<tr>
<td>2.11</td>
<td>MDSD Aproach [Sta06]</td>
<td>22</td>
</tr>
<tr>
<td>2.12</td>
<td>QVT Language Concept [Obj10a]</td>
<td>23</td>
</tr>
<tr>
<td>3.1</td>
<td>Integrating Performance Completions (Source: Happe et al. [HBR+09])</td>
<td>33</td>
</tr>
<tr>
<td>3.2</td>
<td>Example for Coupled Transformations. (Source: Happe et al. [HBR+09])</td>
<td>34</td>
</tr>
<tr>
<td>3.3</td>
<td>Messaging Completion Components. (Source: Happe et al. [HBR+09])</td>
<td>34</td>
</tr>
<tr>
<td>4.1</td>
<td>Approach Overview</td>
<td>37</td>
</tr>
<tr>
<td>5.1</td>
<td>Event Types and Groups</td>
<td>40</td>
</tr>
<tr>
<td>5.2</td>
<td>Roles</td>
<td>41</td>
</tr>
<tr>
<td>5.3</td>
<td>AssemblyEventConnector</td>
<td>42</td>
</tr>
<tr>
<td>5.4</td>
<td>EventDelegationConnectors</td>
<td>42</td>
</tr>
<tr>
<td>5.5</td>
<td>EmitEventAction</td>
<td>43</td>
</tr>
<tr>
<td>5.6</td>
<td>Call Action Class Hierarchy</td>
<td>44</td>
</tr>
<tr>
<td>5.7</td>
<td>RDSEFF for OperationSignature and EventTypes</td>
<td>44</td>
</tr>
<tr>
<td>5.8</td>
<td>Extended Repository Editor</td>
<td>45</td>
</tr>
<tr>
<td>5.9</td>
<td>Example Composite Editor</td>
<td>45</td>
</tr>
<tr>
<td>5.10</td>
<td>UML Activity Diagram - Signal Actions [Obj10b]</td>
<td>46</td>
</tr>
<tr>
<td>5.11</td>
<td>Example SEFF Editor</td>
<td>46</td>
</tr>
<tr>
<td>6.1</td>
<td>Overall Transformation Process</td>
<td>47</td>
</tr>
<tr>
<td>6.2</td>
<td>Source to Intermediate Model Transformation</td>
<td>48</td>
</tr>
<tr>
<td>6.3</td>
<td>Forked Based Event Split</td>
<td>51</td>
</tr>
<tr>
<td>6.4</td>
<td>Sink to Intermediate Model Transformation</td>
<td>51</td>
</tr>
<tr>
<td>6.5</td>
<td>Middleware Model Weaving Alternatives</td>
<td>53</td>
</tr>
<tr>
<td>6.6</td>
<td>Middleware Weaving Process</td>
<td>54</td>
</tr>
<tr>
<td>6.7</td>
<td>Middleware Features respected by the Transformation</td>
<td>55</td>
</tr>
<tr>
<td>7.1</td>
<td>Transformation High-Level Implementation</td>
<td>60</td>
</tr>
<tr>
<td>7.2</td>
<td>Transformation Library Structure</td>
<td>60</td>
</tr>
<tr>
<td>7.3</td>
<td>Common Communication Component Library</td>
<td>63</td>
</tr>
<tr>
<td>7.4</td>
<td>Screenshot Completion Type Selection</td>
<td>68</td>
</tr>
</tbody>
</table>
7.5. Completion Configuration Extension ........................................... 69
7.6. Completion Execution Extension ............................................... 69

8.1. Components of the TIME scenario [RK09b] ...................................... 74
8.2. Evaluation Process ......................................................................... 75
8.3. Source Sink Connection - Old Case Study ....................................... 76
8.4. Source Sink Connection - New Case Study ....................................... 76
8.5. Metric Summary: Number of Elements to Create per Analysed Case ..... 80
8.6. Scenario 1 - CPU Utilisation .......................................................... 82
8.7. Scenario 2 - CPU Utilisation .......................................................... 83
8.8. Scenario 2 - Location Storage Mean Processing Time ....................... 84
8.9. Scenario 3 - CPU Utilisation .......................................................... 84
8.10. Scenario 4 - CPU Utilisation ......................................................... 85
8.11. Exemplary Modelling Efforts to Test New Setups ........................... 87
List of Tables

8.1. Goal Question Metric Plan for the Evaluation ........................................... 73
8.2. Element Creation for an Event-Based Connection ....................................... 78
8.3. Element Creation for an Additional Sink ................................................. 78
8.4. Element Creation for an Additional Source ............................................. 79
8.5. Measured Resource Demands of the Components (in ms) ............................... 81
8.6. Effort Removing Single Threaded Characteristic ...................................... 86
8.7. Replace Middleware ................................................................................. 86
A.1. Scenario 1 Results ..................................................................................... 94
A.2. Scenario 2 Results Server 1 ........................................................................ 95
A.3. Scenario 2 Results Server 2 ........................................................................ 96
A.4. Scenario 3 Results ..................................................................................... 97
A.5. Scenario 4 Results ..................................................................................... 98
Bibliography


[eP02] P. N. e.V. PNO, “Profibus technology and application system description,” PROFIBUS Nutzerorganisation e.V. PNO, October 2002.


