Journal on Software and Systems Modeling manuscript No.
(will be inserted by the editor)

Modeling Event-based Communication in Component-based Software Architectures for Performance Predictions

Christoph Rathfelder[1], Benjamin Klatt[1], Kai Sachs[2], Samuel Kounev[3⋆]

[1] FZI Research Center for Information Technology
Karlsruhe, Germany
{rathfelder; klatt}@fzi.de
[2] SAP AG
Walldorf, Germany
kai.sachs@sap.com
[3] Karlsruhe Institute of Technology (KIT)
Karlsruhe, Germany
kounev@kit.edu

Received: date / Revised version: date

Abstract Event-based communication is used in different domains including telecommunications, transportation, and business information systems to build scalable distributed systems. Such systems typically have stringent requirements for performance and scalability as they provide business and mission critical services. While the use of event-based communication enables loosely-coupled interactions between components and leads to improved system scalability, it makes it much harder for developers to estimate the system's behavior and performance under load due to the decoupling of components and control flow. In this paper, we present our approach enabling the modeling and performance prediction of event-based systems at the architecture level. Applying a model-to-model transformation, our approach integrates platform-specific performance influences of the underlying middleware while enabling the use of different existing analytical and simulation-based prediction techniques. In summary the contributions of this paper are: i) the development of a meta-model for event-based communication at the architecture level, ii) a platform aware model-to-model transformation, and iii) a detailed evaluation of the applicability of our approach based on two representative real-world case studies. The results demonstrate the effectiveness, practicability and accuracy of the proposed modeling and prediction approach.

⋆ This work was partially funded by the German Research Foundation (grant No. KO 3445/6-1)

1 Introduction

The event-based communication paradigm is used increasingly often to build loosely-coupled distributed systems in many different industry domains. The application areas of event-based systems range from distributed sensor-based systems up to large-scale business information systems [24]. Compared to synchronous communication using, for example, remote procedure calls (RPC), event-based communication among components promises several benefits such as high scalability and extendability [25]. Being asynchronous in nature, it allows a *send-and-forget* approach, i.e., a component that sends a message can continue its execution without waiting for the receiver to react on the message. Furthermore, the loose coupling of components achieved by the mediating middleware system leads to an increased extensibility of the system as components can easily be added, removed, or substituted.

With the growing proliferation of event-based communication in mission critical systems, the performance and scalability of such systems are becoming a major concern. To ensure adequate Quality-of-Service (QoS), it is essential that applications are subjected to a rigorous performance and scalability analysis as part of the software development process. In today's data centers, software systems are often deployed on server machines with over-provisioned capacity in order to guarantee highly available and responsive operation [28], which automatically leads to lower efficiency. Although the event-based communication model promises to improve flexibility and scalability, the complexity compared to direct RPC-based communication is higher since the application logic is distributed among multiple independent event handlers with decoupled and parallel execution paths. This in-

creases the difficulty of modeling event-based communication for quality predictions at system design and deployment time. Thus, the evaluation of event-based systems requires specialized techniques that consider the different characteristics and features of event-based communication.

Performance modeling and prediction techniques for component-based systems, surveyed in [33], support the architect in evaluating the system architecture and design alternatives regarding their performance and resource efficiency. However, most general-purpose performance meta-models for component-based systems provide limited support for modeling event-based communication. Furthermore, existing performance prediction techniques specialized for event-based systems (e.g., [42, 9, 54]) are focused on modeling the routing of events in the system as opposed to modeling the interactions and message flows between the communicating components. As an example of a representative mature meta-model for component-based software architectures, we consider the Palladio Component Model (PCM) [6]. PCM is accompanied with different analytical and simulation-based prediction techniques, e.g., [6, 38, 34] enabling quality predictions at system design time. Similarly to most component meta-models for component-based architectures, PCM, in its original version did not provide support for modeling of event-based communication. Performance predictions are only possible using workarounds as demonstrated in [48]. The modeling effort incurred by this manual workaround approach is very high and provides limited flexibility to evaluate different design alternatives. In [50], we briefly sketched the core elements required for modeling event-based communication. In a follow up poster paper [49], we described our idea of using a model-to-model transformation to map the newly introduced model elements to existing PCM model elements allowing to use the available prediction techniques, while significantly reducing the modeling effort. In [29], we presented an extension of the PCM combined with an initial implementation of such a model-to-model transformation that was limited to modeling point-to-point connections between components.

In this paper, we present our integrated approach enabling the comprehensive modeling and performance prediction of event-based communication as part of architecture-level models. We extend our work presented in [29] to additionally support the modeling and prediction of publish/subscribe systems and thus extends the scope of our approach to a new domain of event-based systems. The implementation of our approach is based on PCM as a mature and representative meta-model for architecture-level performance predictions. The described meta-model elements combined with the presented model-to-model transformation cover point-to-point as well as publish/ subscribe communication and thus the most often used communication styles in event-based systems. Furthermore, this paper presents a detailed evaluation of our approach based on two real-world case studies representing different domains and communication styles of event-based systems.

The contributions of the paper are i) the identification and implementation of meta-model elements required for modeling event-based communication at the architecture level, ii) the design and realization of the two-step model-to-model transformation integrating platform-independent and platform-specific aspects of event-based communication into the prediction model, and finally iii) a detailed evaluation of the presented approach based on two representative real-world case studies.

Modeling event-based communication Modeling event-based communication at the architecture level requires new meta-model elements. We extended our work in [29], which was limited to direct point-to-point connections with elements enabling the modeling of publish/subscribe communication in component-based systems. Theses extensions open up a new application domain of industrial systems for our approach. We implemented these meta-model extensions based on PCM, as a mature and representative meta-model for component-based architectures.

Platform-aware refinement transformation We developed a two-step model-to-model transformation responsible for integrating the performance relevant influence factors of event-based communication into the prediction model. The first step refines the event-based point-to-point and publish/subscribe communication links with a detailed event-processing chain while the second steps integrates platform-specific components. The clear separation of platform-independent and platform-specific aspects supports architects in evaluating the influence of different middleware implementations on the system performance.

Evaluation In order to provide a comprehensive evaluation of our approach we selected two representative real-world systems from different application domains and covering most aspects of event-based systems. The first case study is based on a traffic-monitoring system built on top of the distributed peer-to-peer middleware SBUS [27]. In addition to the prediction accuracy, we evaluated the adaptability of our prediction model to reflect architectural changes and deployment options, which are typical for distributed event-based systems. The SPECjms-2007 benchmark[1], our second case study, is a supply chain management system representative for real-world industrial applications built on top of a central-

[1] SPECjms2007 is a trademark of the Standard Performance Evaluation Corporation (SPEC). The results or findings in this publication have not been reviewed or accepted by SPEC, therefore no comparison nor performance inference can be made against any published SPEC result. The official web site for SPECjms2007 is located at http://www.spec.org/osg/jms2007.

ized message-oriented middleware. The different interactions exercise a complex transaction mix including point-to-point and publish/subscribe communication [52]. The results of our evaluation demonstrate that our approach reduces the modeling efforts by more than 80% compared to the use of workarounds. The effort to reflect different design alternative and multiple deployment options in the architecture level models is less than 30 minutes. In both case studies, the detailed evaluation of the prediction accuracy shows that the prediction error, compared to measurements on the running system, is less than 20% in most cases. This is a more than acceptable accuracy for design-time performance analysis [39].

The remainder of this paper is organized as follows. Section 2 presents the foundations of our work and introduces event-based systems and software performance engineering in general. Additionally, it presents an overview on PCM which is the basis of our implementation. Section 3 presents the meta-model elements enabling the modeling of event-based communication at the architecture level. Section 4 describes the model-to-model transformation and the separation of platform-specific and platform-independent aspects. Section 5 presents a detailed evaluation of our approach in the context of two representative real-world case studies. Finally, in Section 6, we give an overview of related work and conclude with a brief summary and a discussion of ongoing and future work in Section 7.

2 Foundations

In the following, we present an overview on event-based systems and event-based communication. Furthermore, we introduce software performance engineering in general and PCM in particular, which we selected as basis for our implementation as it is a mature and representative meta-model for performance predictions of component-based systems.

2.1 Event-based Systems

Event-based systems are used in a variety of different domains and their size ranges from small embedded up to large-scale and world-wide distributed systems [24]. Nevertheless, all event-based systems have four core elements in common: *Events*, *Sources*, *Sinks*, and the *Middleware* [8]. Events are data elements asynchronously transferred between components to trigger a certain behavior or to transfer data. They are instantiated within sources, which are responsible to publish and emit the event. The counterpart of a source is the sink, which receives and processes events. The communication between sources and sinks is enabled by a communication middleware supporting loosely-coupled communication among distributed software components. This allows a source to send an event and then continue working while the event is being delivered and processed.

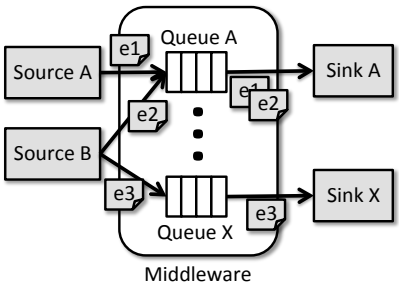


Fig. 1 Point-to-point Communication

Although all modern event-based systems are built using a middleware system, the implementation and architecture can significantly differ. Using a *centralized* approach, the middleware is running as one central instance all sources and sinks are connected to. Most of the event-based middleware platforms currently used in industry (e.g., IBM WebSphere MQ, TIBCO EMS) support the Java Message Service (JMS) [56] standard interface for accessing a centralized server or server cluster. In *peer-to-peer* systems the middleware is integrated into the sources and sinks as local libraries without a dedicated server or set of servers hosting the middleware.

In event-based systems, two communication styles, which are independent of the middleware's architecture, need to be differentiated. In *point-to-point* communication (see Figure 1), the events are sent to a specified queue associated with exactly one sink. Due to the single queue, the interaction type is limited to many-to-one communication. In contrast,architecture level in *publish/subscribe* systems, a sink connects to the middleware system and subscribes for the events of interest [14]. As illustrated in Figure 2, the middleware provides different event channels that are used to group the events and thus simplify the subscription, as sinks only have to connect to an event channel. These channels can represent a certain topic (*topic-based subscription*) or a certain type of events (*type-based subscription*). Furthermore, the subscription can include individual filtering rules applied to the content of an event (*content-based subscription*). When emitting an event, the source is re-

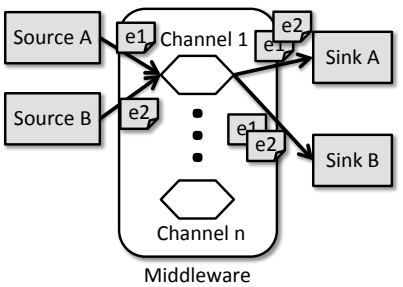


Fig. 2 Publish/subscribe Communication

sponsible for selecting the corresponding channel that is used to publish the event. The middleware forwards the event to all sinks subscribed to this channel. For a detailed introduction to event-based systems the reader is referred to [25,51,41,23].

2.2 Software Performance Engineering

Over the last fifteen years, a number of approaches have been proposed for integrating performance prediction techniques into the software engineering process. Efforts were initiated with Smith's seminal work on Software Performance Engineering (SPE) [55]. Since then, a number of architecture-level performance meta-models have been developed by the performance engineering community. The most prominent examples are the UML SPT profile [44] and its successor, the UML MARTE profile [45]. Both of them are extensions of the UML as the de-facto standard modeling language for software architectures. All approaches support the evaluation of system performance at design time and the comparison of different design alternatives and deployment options.

The OMG has defined a general process for model-based performance predictions [5]. The starting point of this process is a model that describes the software system itself using an established modeling language, such as the UML. Those software models do not include any specific information regarding the performance characteristics of the modeled system, e.g., resource demands and parameter dependencies. This information is added in a second step. If an implementation is already available measurements of the system 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 KLAPER [18] or PCM [21]. The annotated software model is used as input for a transformation to a stochastic performance model such as a layered queueing network (LQN) or a queueing Petri net (QPN). The stochastic performance model is then evaluated with analysis or simulation techniques. In a final step, the prediction results are returned as a feedback related to the original software model. A recent survey of performance engineering models and techniques focusing on component-based systems was published in [33].

2.3 Palladio Component Model

The Palladio Component Model (PCM) [21] is a domain-specific modeling language for component-based software architectures. It supports an automated transformation of architecture-level performance models to predictive performance models including LQNs [34], QPNs [38] and simulation models [5]. PCM supports the evaluation of different performance metrics, including response time, maximum throughput, and resource utilization.

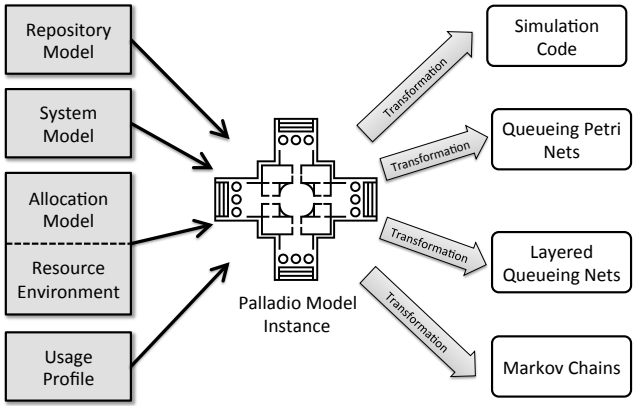


Fig. 3 Palladio Overview

The PCM approach provides an Eclipse-based modeling and prediction tool [2]. Further details and a technical description can be found in [6].

The performance of a component-based software system is influenced by four factors [5]: The implementation of system components, the performance of external components used, the deployment platform (e.g., hardware, middleware, networks), and the system's usage profile. In PCM, all factors are modeled in specific sub-models and thus can be managed independently. The composition of these models forms a PCM instance.

The *Repository Model* specifies a library of system components and their behavior. Components provide and require interfaces. PCM provides a description language, called *ResourceDemandingServiceEffectSpecification* (RDSEFF) to specify the behavior of a component including the resource demands and parameter dependencies. This language provides actions to model the internal control flow, such as Branch- or LoopActions, but also to call other components through the required interfaces, which are called ExternalCallActions. For a detailed description we refer to [6]. The *System Model* describes the structure of the system by instantiating and connecting components defined within the *Repository* via their provided and required interfaces. In the *Allocation Model*, the components defined within the *System Model* are allocated to physical resources described in the *Resource-Environment Model*. The model specifies the hardware environment the system is executed on, e.g., servers, processor speed, network links. The *Usage Model* describes the workload induced by the system's end-users. The *Usage Model* describes the behavior of users and their invocation of interfaces provided by the system. For example, the workload may specify how many users access the system, their input parameters, and the inter-arrival time of requests. Usage profiles within the model represent individual user behaviors. Usually, parameter values have an influence on the software's behavior and the resource demands. However, it is often not possible to explicitly model these dependencies. With StochasticExpressions (StoEx), PCM offers a language to describe direct pa-

rameter dependencies based on boolean and mathematical operations, but also multiple probability distribution functions to abstract complex or unknown parameter dependencies.

As illustrated in Figure 3, the combination of the previously described models builds an instance of a PCM model, which then can be transformed into multiple prediction techniques. The transformation and the execution of the prediction is encapsulated within the Palladio tool and transparent for the software architect. The prediction results, e.g., resource utilization, response times or throughput for individual components as well as the whole system are visualized and returned to the architect.

3 Modeling Event-based Communication

Modeling event-based communication is a neglected factor in most architecture-level performance models. Although some of them already support to specify asynchronous method calls [33], the queuing effects and the specification of publish/subscribe systems is not supported. In the following, we exemplarily describe the meta-model abstractions required for modeling event-based communication at the architecture level using PCM as a representative meta-model of component-based systems. Although, we demonstrate our extensions in the context of PCM, as one of the most advanced modeling and prediction tools, the general approach is not limited to PCM and can, with slight adaptations, be applied to most modeling and performance prediction approach for component-based systems.

In [29], we have already described the elements required for modeling events, sink and source roles of components, as well as direct connectors between components. However, to keep this paper self-contained, we first give an overview on these elements before presenting the new extensions enabling the modeling of publish/subscribe interactions between components and the specification of sink-specific filtering rules.

3.1 Interfaces and Events

In component-based systems, interfaces describe the contract between two components. In synchronous RPC-style communications, interfaces, called `OperationInterface`, combine a set of `OperationSignatures`. These signatures describe operations required by one component and provided by another. In event-based communication, the contract does not describe a set of operations that can be called but rather the event types a component can emit or receive and process. To describe the individual events produced or consumed by a component, we specify a meta-model element named `EventType`. Events often contain a payload that can be a simple value as well as a complex data type. To

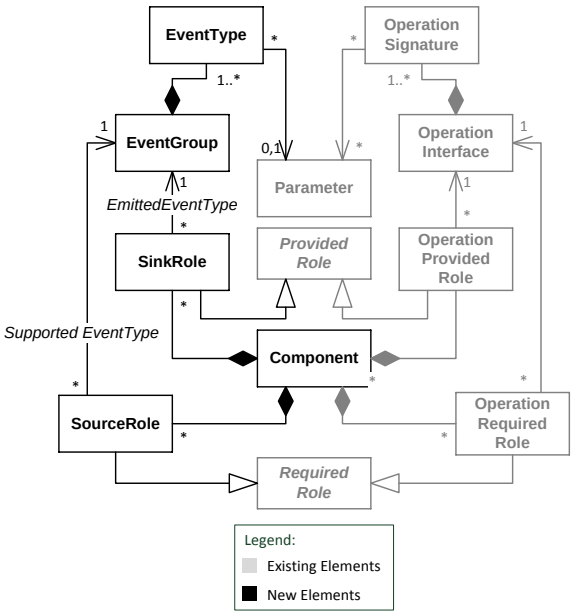


Fig. 4 Component and Roles

enable modeling of an event's payload, the `EventType` contains a `Parameter`, which is a PCM element used to specify the performance-relevant characteristics of simple or complex datatypes. Furthermore, we introduce the `EventGroup` as a specialization of the abstract `Interface` to group `EventTypes`.

3.2 Sink and Source Roles

Our approach is aligned with the general PCM concept for providing and requiring component functionality. The PCM meta-model contains an abstract `ProvidedRole` and an abstract `RequiredRole` element, which describe the roles of a component within a component connection. A component receiving events provides functionality to process them. Thus the `SinkRole` is a specialized `ProvidedRole`. As described in Section 3.3, each `SinkRole` is connected with a dedicated event handler specification and references the `EventGroup` and thus `EventTypes` that can be processed. The counterpart is the `SourceRole`, which is a specialization of a `RequiredRole`. It refers the component able to emit events of the types described by the referenced `EventGroup`. Figure 4 illustrates the different roles a component can contain.

3.3 Behavior

The meta-model extensions, we presented above, cover only the static aspects of a component. In order to reflect the behavioral aspects, we define elements reflecting the creation and publishing of events as well as their processing.

In PCM, the behavior of a component is modeled with RDSEFFs as described in Section 2.3. We defined

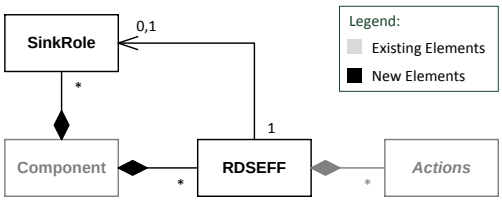


Fig. 5 Event Processing

a new element representing the instantiation of an event as part of the behavior description, named `EmitEventAction`. It references the `SourceRole` of the component that should be used to publish the event. Additionally, it includes a `VariableCharacterization` used to specify the event's value and its performance relevant characteristics. In PCM, `VariableCharacterizations` are used to specify the instantiation and the value assignment to variables and parameters.

To model and specify the event processing within a component, we introduce a concept analogue to the specification of provided functionality based on `OperationalInterfaces`. As illustrated in Figure 5, a component includes for each `SinkRole` a `RDSEFF` specifying the processing of events received through this role.

3.4 Event Channels

The publish/subscribe communication, often used in event-based systems, introduces a higher decoupling of components [14]. In order to support point-to-point as well as publish/subscribe communication, we defined a new meta-model element named `EventChannel`. Channels enable the many-to-many communication as sources and sinks can independently be connected to a channel. An `EventChannel` refers to an `EventGroup` and all connected `Sink-` and `SourceRoles` must support this `EventGroup` which ensures type-safe processing of events. The `EventChannel` is part of a PCM *System Model* and thus can be explicitly deployed on a dedicated or shared resource. In addition to the decoupling, the event channels used in publish/subscribe (see Figure 2) can also be used to structure the system at the architecture level by grouping events from different sources into one channel instead of several direct connections from sources to sinks.

3.5 Connectors

In the *System Model*, the architect defines the system architecture by instantiating components and connecting their provided and required interfaces respectively sources and sinks of a component. In PCM, component instances are called `AssemblyContexts`.

In order to support event-based communication, we specified new component connectors. In the existing operational case, only one-to-one connections were allowed. In the new event-based scenario, sinks are able to handle events that are emitted by one or more sources and the events of one source can be received and processed by zero, one or many sinks in parallel.

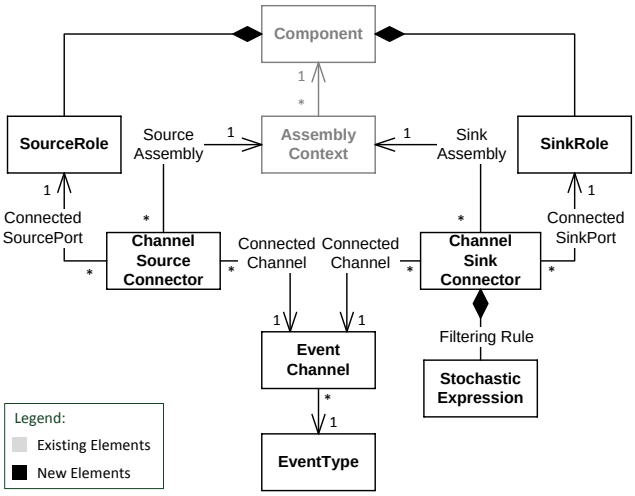


Fig. 6 Event Channel and Connectors

In addition to the `AssemblyEventConnector` describing direct point-to-point connections, we define two new types of connectors for modeling publish/subscribe communication, namely `ChannelSourceConnectors` and `ChannelSinkConnectors`, which are illustrated in Figure 6. Both connectors refer to an `EventChannel`, to an `AssemblyContext`, and to the belonging `SourceRole` respectively `SinkRole`.

To specify sink-specific filtering rules, we extended the `AssemblyEventConnector` and the `ChannelSinkConnectors` with an additional `StochasticExpression` based on PCM's StoEx language as `FilteringRule`. In addition to value-based filtering rules like "`event.BYTESIZE <= 1000`", which filters out large messages, or "`event.TYPE == ERROR`", which selects only error messages, PCMs StoEx language supports probabilistic expressions, e.g., 80% of the generated events should be forwarded to the sink. Probabilistic filters enable modeling unreliable event processing as well as abstracting from concrete value dependencies or load balancing strategies.

4 Platform-Aware Refinement Transformation

The meta-model elements introduced in the previous section enable software architects to model event-based communication in component-based systems at the architecture-level. The extensions allow an explicit modeling of one-to-many, many-to-one as well as many-to-many interactions between components while abstracting platform-specific details of the underlying communication middleware. This abstraction of communication and implementation details is aligned with the notion of

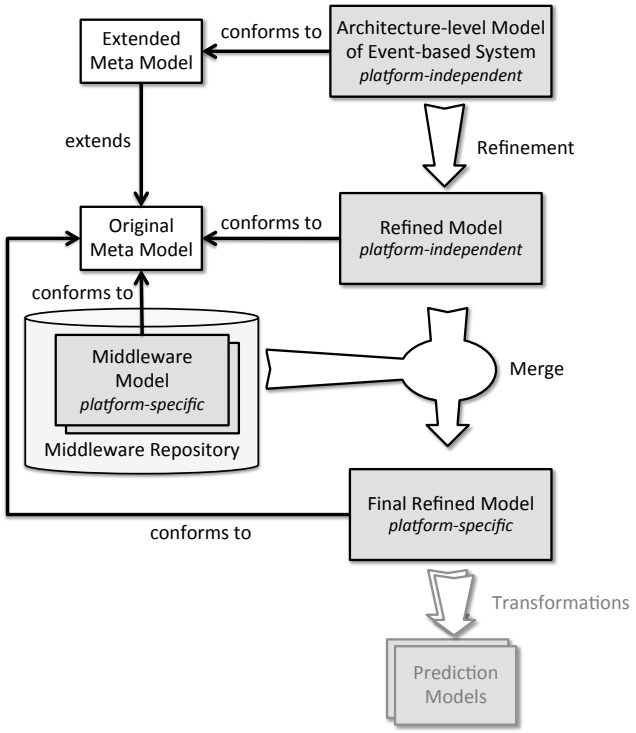


Fig. 7 Transformation Overview

a platform-independent model (PIM) as defined by the OMG [43].

In order to derive a platform-specific model (PSM) that contains platform-specific details about the behavior of the underlying communication middleware, the architecture model is refined by applying a two-step transformation as depicted in Figure 7. First, a platform-independent event processing chain is integrated. This refinement step substitutes the new meta-model elements with several components representing the different event processing steps executed inside the transmission system. The next step of the transformation integrates middleware-specific components specified in a dedicated middleware model capturing the performance relevant influence factors of the employed communication middleware. Since the middleware models are independent of the system architecture model, they are stored in a middleware repository and can be reused in multiple system evaluations. The resulting model serves as input for all existing prediction techniques, available for the original version of PCM, as all new elements are substituted.

In the following, we first describe the generic event processing chain that provides a skeleton to integrate platform-specific components representing the different event processing activities within the transmission system. Second, we provide an illustration of the two-step transformation explaining the refinement into the behavior-equivalent model as well as the merging with the middleware model. For a detailed description and formalization of the transformation we refer to [47].

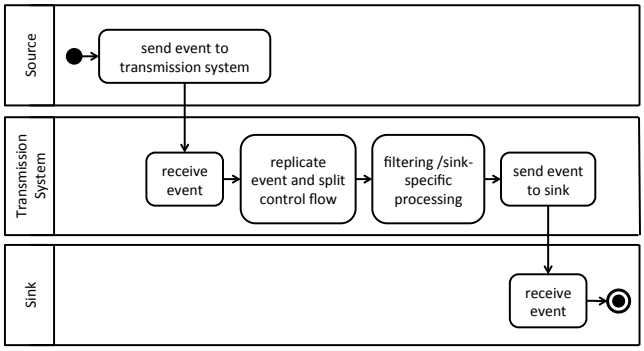


Fig. 8 Generic Event Processing Chain

4.1 Event Processing Chain

The generic event processing chain, illustrated in Figure 8, consists of six processing stages that are common for event-based systems. The execution of the different stages is distributed among the involved source and sink components and the transmission system. Given that the processing chain is defined to be platform-independent, it does not include any concrete resource demanding behavior, however, it provides placeholders to integrate such platform-specific behavior that is executed within the various stages.

The first stage, *send event to transmission system*, is performed on the source-side and includes the communication activities to hand-over the event to the transmission system. This stage is usually performed within a local library, which encapsulates the communication and includes activities like marshaling, compression, or encryption on the source-side. In the parallel *receive event* stage, the event is received by the transmission system, which includes the communication with the source component and possibly additional activities such as the demarshaling required to acknowledge the correct receipt of the event.

Asynchronous many-to-many communication between components is one of the main characteristics of event-based interactions. In the generic event-processing chain, this behavior is reflected by the *replicate event and split control flow* processing stage. While providing a cloned instance of the event to each connected sink, the control flow between sources and sinks is decoupled and the cloned events are forwarded to the sinks in parallel. The remaining activities of the event processing chain are executed in parallel and independently for each connected sink.

After splitting the control flow, the generic event processing chain contains the *sink-specific filtering* based on the filtering conditions defined within the connectors. If the event matches the defined filtering conditions for a given sink, the event is further processed. Otherwise, the event processing for the respective sink is terminated. In addition to the filtering, which is considered as platform-independent logic, the filtering stage allows to

integrate additional platform-specific processing such as data conversion, deserialization, or decompression. Such platform-specific activities are described as part of the middleware model which is later integrated when deriving the final platform-specific model.

Similarly to the communication between the source and the transmission system, which is reflected by the first two stages of the event processing chain, the communication between the transmission system and the sinks is split into two stages. The *send event to sink* stage encapsulates the communication aspects the transmission system is responsible for while allowing the integration of platform-specific marshaling or serialization operations. The *receive event* stage is the counterpart stage on the sink-side usually executed in parallel by a local library encapsulating the communication with the transmission system.

The presented platform-independent event-processing chain is the foundation for the platform-independent refinement transformation presented in the following section.

4.2 Platform-independent Refinement

The platform-independent refinement, which is the first step of our two-step transformation, substitutes the event-based interactions modeled at the architecture-level with a chain of components. Each of these components is responsible for exactly one of the presented processing stages. In the following, we present the transformation of source and sink components to illustrate our approach.

Point-to-point and publish/subscribe interactions are modeled differently at the architecture level. In the first case, direct connectors between sources and sinks are used while in the second case sources and sinks are connected through intermediary event channels. In the following, we first give an overview on the refinement of direct point-to-point connectors followed by an explanation of the differences in case of publish/subscribe connections.

4.3 Refinement of Point-to-Point Connectors

Figure 9 presents an illustration of the refinement of a source component connected with two sink components using direct point-to-point connectors. The `SourceRole` as part of component `A` is replaced by a required operation interface resulting in a synchronous call initiating the event processing chain. This interface is connected with the provided operational interface of the newly introduced `SourcePort` component, which represents the local library encapsulating the communication with the transmission system as part of the *send event to transmission system* stage. The `SourcePort` is always deployed on the same node as the source component itself.

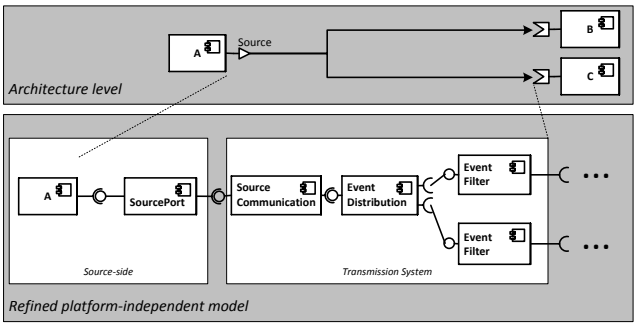


Fig. 9 Refinement of a Source with Point-to-Point Connectors

The `SourceCommunication` component inside the transmission system receives the emitted event from the `SourcePort` component. The `SourceCommunication` component provides a skeleton to integrate platform specific-components describing the resource demands for receiving and processing the event.

The `EventDistribution` component is responsible for replicating the event and splitting the control flow for each connected sink. The component contains an individual `OperationRequiredRole` for each connected sink. To realize the asynchronous and decoupled behavior of event-based communication, the behavior description of the `EventDistribution` component makes use of an asynchronous fork. Each `ForkBehaviour` includes an `ExternalCallAction` associated with one of the newly added required interfaces. As the number of required interfaces and forks depends on the number of connected sinks, the `EventDistribution` component is individually generated for each source.

Each of the required interfaces is connected with a sink-specific `EventFilter` component. In contrast to the other components, which directly call the next component in the chain of responsibility, the `EventFilter` components include a `BranchAction` to call the next component only if the filtering rule defined as a stochastic expression (StoEx) evaluates to true. Otherwise, the event processing for this specific sink is terminated.

Similarly to the transformation of sources, each SinkRole is replaced with a provided operational interface and two additional components as illustrated in Figure 10. `SinkCommunication` is the first component in the event processing chain resulting from this refinement and it provides a skeleton to integrate relevant resource

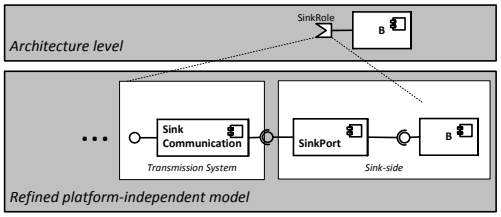


Fig. 10 Transformation of a Sink

demanding behavior of the transmission system when communicating with the respective sink. The counterpart of `SinkCommunication` is the `SinkPort` component abstracting the local library of the sink component and its local resource demanding behavior at the sink side. In addition to the introduced provided operational interface, the sink component is modified to handle the incoming operation calls of the transmission system when events are delivered. The existing RDSEFFs of the component are linked to the respective `Signatures` of the `OperationalInterfaces` they are defined for.

To support peer-to-peer-based as well as centralized middleware systems, the components representing the transmission system are deployed differently depending on the specification of a central `ResourceContainer` hosting the transmission system. If the `ResourceEnvironment` contains such a specification, the components are deployed on this node otherwise they are deployed on the resource container hosting the source component.

4.4 Refinement of Event Channels

As shown in Figure 11, the transformation of publish/subscribe connections using `EventChannels` is quite similar to point-to-point connections and differs only in the explicit deployment of channels to dedicated `ResourceContainers`. For each source, the transformation generates an instance of a *SourcePort* component deployed on the same `ResourceContainer` as the respective source, while the middleware components are instantiated once per `EventChannel` and deployed on the respective `ResourceContainer` associated with the channel.

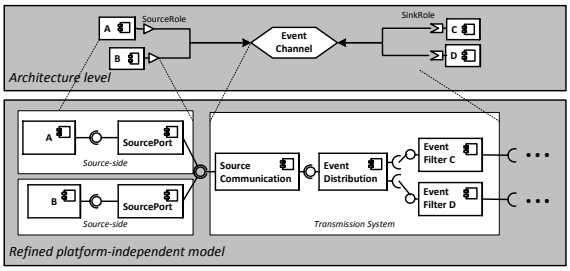


Fig. 11 Refinement of Event Channels

4.5 Merging with Platform-specific Middleware Components

From a modeling point of view, the general event-based connections between components and the specific middleware used for the technical implementation are at two different levels of abstraction. For this reasons, we separated the platform-specific behavior and resource demands of a middleware implementation using a separate middleware repository. The middleware repository,

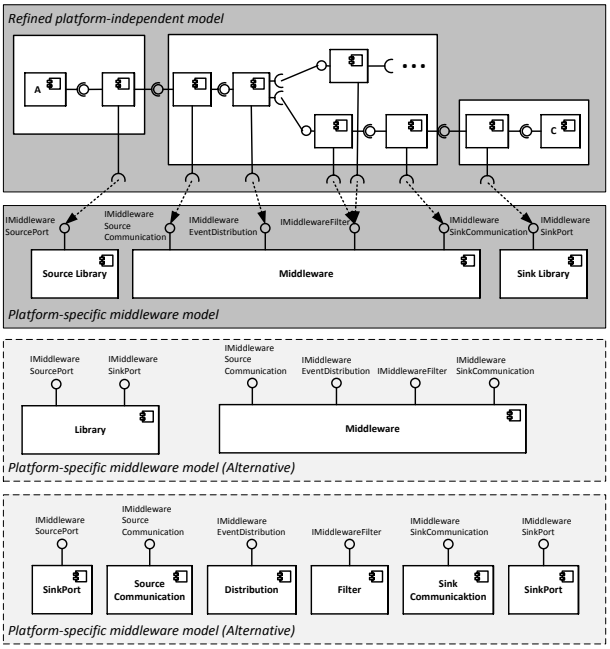


Fig. 12 Examples of Middleware Models and their Weaving

which is also based on PCM, includes six predefined operational interfaces, namely `IMiddlewareSourcePort`, `IMiddlewareSourceCommunication`, `IMiddlewareEventDistribution`, `IMiddlewareFilter`, `IMiddlewareSinkCommunication` and `IMiddlewareSinkPort`. The middleware repository can contain a dedicated component for each interface but also allows to specify only one component providing all interfaces. Figure 12 illustrates possible exemplary alternatives.

The integration of the platform-specific components into the platform-independent processing chain consists of several steps. The first step is the identification and localization of components providing one or more of the middleware interfaces. Second, the components representing the platform-independent event processing are extended to invoke the corresponding middleware component. As third step, the deployment of the middleware components is generated. They are deployed on the same resource container as the respective platform-independent component. The transformation ensures that each resource container contains at most one instance of each middleware component, which is shared between the platform-independent components. Finally, the merging transformation generates the connectors between the new required interfaces of the event processing components with the provided interfaces of the middleware component instance on the same resource container. The result of the model merging is the refined platform-specific model that serves as input for different existing analysis and prediction techniques defined for the original PCM.

4.6 Implementation

To perform the evaluation described in the following, we implemented our approach as part of the Palladio tool. We applied the presented model extension to the Ecore-based PCM meta-model. Based on the meta-model, we used the capabilities of the Exlipse Modeling Framework (EMF) to generate basic tree editors and and the code for manipulating model instances. Although, these editors provide functionality to create and change model instances, they are not useable to build models of realistic systems with multiple connections and references. For this reason, we extended the graphical editors of the Palladio tool using the Eclipse Graphical Modeling Framework (GMF) to support the modeling of event-based communication as shown in Figure 13. The refinement model-to-model transformation is implemented with the transformation language QVTO. We integrated the transformation into the prediction workflow. It is automatically executed if event-based communication is used in the model. Finally, we extended the performance prediction dialog to allow the selection of the middleware repository.

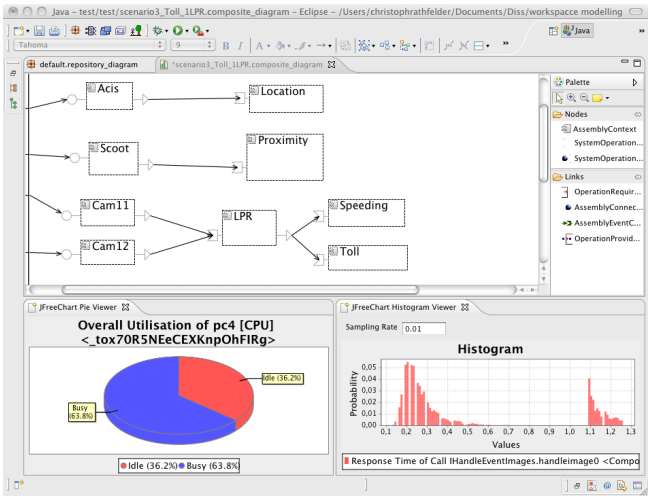


Fig. 13 Screenshot of the Palladio Tool

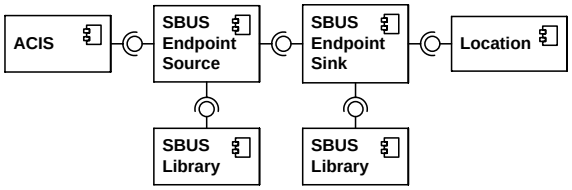
5 Evaluation

The goal of our approach is to i) improve the modeling of event-based communication at the architecture level and ii) enable the performance prediction using architecture-level models. To validate the applicability of our approach we first analyzed the achieved effort reduction of our approach compared to the use of the original PCM with modeling workarounds as shown in [48]. Second, we applied our approach to evaluate the performance of different design alternatives and deployment options and evaluated the effort required to adapt the models. To evaluate the prediction accuracy and the applicability to different software domains, we selected two real-world systems representing different types and domains of event-based systems. The first case study is based on the traffic monitoring system developed within the TIME research project [3] to monitor the traffic in the City of Cambridge. It is based on a decentralized peer-to-peer middleware SBUS [27]. As a representative application for business information systems based on a centralized middleware with point-to-point as well as publish/subscribe communication, we selected the SPECjms2007 benchmark [52].

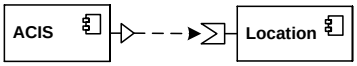
5.1 Reduced Modeling Effort

The original PCM meta-model did not provide any elements specific to events and event-based communication. As shown in a previous case study [50] based on a subset of the traffic monitoring system, it is possible to conduct PCM-based performance predictions using a set of workarounds. These workarounds enabled the architect to setup performance equivalent structures. However, their modeling is very time consuming and they lead to a semantically incorrect model as they are based on synchronous interfaces combined with forks to emulate asynchronous behavior. With the presented approach, we introduce new meta-model elements for explicit modeling of events, source and sink roles, event channels as well as related connectors. While there is no quantitative quality index for such a difference, it can be clearly stated that there is an improvement on the coverage of event-related elements. Figures 14(a) and 14(b) present an event-based point-to-point connection using the workaround and respectively using our meta-model extension. Although, they are performance equivalent, the semantics of the models are different. Using the workaround-based synchronous interfaces combined with forks, at the architecture-level event-based and RPC-style connection can not be distinguished. In contrast, the new elements allow an architect to explicitly differentiate between synchronous call-return behavior and the fire-and-forget behavior of event-based communication.

In addition to enabling the semantically correct modeling of event-based communication, the new elements significantly reduce the modeling effort. We tracked the effort for three different scenarios. In the first scenario, a new sink is added to an existing connection while in the second one a new source is added. In the third scenario, a completely new point-to-point connection between a source and a sink is created. We tracked the effort in terms of the number of elements that must be created without considering the time required for the creation of the individual elements. This was done to avoid the influence of the individual experience and training of the architects in the usage of Eclipse modeling tools in general and the Palladio tools in particular. Adding an additional sink was reduced to create



(a) Source to Sink Connection using the Workarrounds



(b) Source to Sink Connection using the Meta-Model Extension

Fig. 14 Comparison of Source and Sink Modeling

only one element instead of 14 with the old approach (effort reduction 92.8%). The manual modeling reuses existing components as much as possible, but as already mention in Section 4.2, adding a new sink requires for example the extension of the component splitting the control-flow with an additional required interface and the included behavioral specification with an additional fork. The effort for adding an additional source was reduced from 35 to 6 elements (effort reduction 82.8%). Most of the manual effort was required for specifying the `VariableUsages` and `VariableCharacterisations` that forward the event's content through the different components. For a completely new connection, the required effort was reduced from 59 to only 11 elements with the new approach, which is an effort reduction of 81.3%.

Change Scenario	New	Workaround	Effort
	# elem.	# elem.	Reduction
Add Sink	1	14	92,8%
Add Source	6	35	82,8%
New Connection	11	59	81,3%

Table 1 Reduction of Modeling Effort

The results listed in Table 1 are clear indicators for the reduced effort to model event-based communication between components. Even without measuring the time required per element creation in an empirical study, the results of the metric highlight the benefits of the new approach.

5.2 Accuracy of Prediction Results

Beside the effort required to create the performance models for a specific system, the accuracy of the prediction results is one of the most important characteristics of model-based performance predictions. In the following, we introduce two real-world case studies, representing different domains and communication types. The first case study is using a distributed traffic monitoring system based on a decentralized middleware with point-to-point communications only. In addition to the prediction accuracy, we also present the variability and adaptability of the models conducting performance prediction for different system evolution stages and design alternatives. The second case study is based on the SPECjms2007 benchmark using a centralized middleware with point-to-point as well as publish/subscribe communication.

5.2.1 Traffic Monitoring Case Study The system under study is a traffic monitoring application based on results of the TIME project (Transport Information Monitoring Environment) [3] at the University of Cambridge. It consists of several components emitting and consuming different types of events. The system is based on the novel component-based middleware SBUS, which was also developed as part of the TIME project. The SBUS framework encapsulates the communication between components and thus enables easy reconfiguration of component connections and deployment options without affecting the component's implementation. After a short introduction of SBUS, we present the different components the traffic monitoring system consists of. Finally, we apply our approach and demonstrate the evaluation of different design alternatives and deployment options. For each of the three scenarios, we describe the required adaptations of the architecture-level models and compare the predicted performance metrics with measurements conducted in our testbed.

SBUS middleware The SBUS middleware is based on a peer-to-peer architecture and supports point-to-point event-based communication including continuous streams of data (e.g., from sensors), asynchronous events, and synchronous remote procedure calls (RPC). In SBUS, each component is as illustrated in Figure 15 divided into a wrapper, provided by the SBUS framework, and

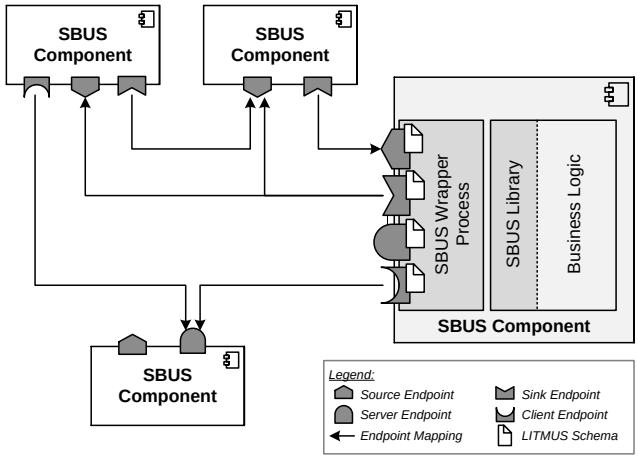


Fig. 15 Schematic Overview on SBUS Components

the component's functionality itself. The wrapper manages all communication between components, including handling the network, registration of event sinks and sources, and marshaling of data. With SBUS, sources and sinks can be connected during run-time without any influences on the component's internal implementation. This allows to build highly adaptable systems which can be extended and adapted at run-time. The basic entity of SBUS is the component, which can define multiple communication endpoints. Each endpoint can be a client, a server, a source, or a sink port. Clients and servers implement RPC functionality, providing synchronous request/reply communication, and are attached in many-to-one relationships. On the other hand, streams of events emitted from source endpoints are received by sink endpoints in a many-to-many fashion.

Thanks to the SBUS middleware, which completely encapsulates the communication between components, the deployment of components as well as their connections can be changed with almost no effort. However, the influence of such changes on the system performance are hard to estimate.

Traffic Monitoring Application The traffic monitoring application we studied consists of 8 different types of SBUS components (see Figure 16).

As described in [15], street lamps are equipped with cameras. These cameras only collect anonymized statistical data. We extended this scenario with cameras that take pictures of each vehicle on the street. Each of these cameras is equipped with an SBUS component (the *Cam* component), which is responsible to send the picture combined with position information of the camera and a timestamp in the form of an event to all connected sinks.

The *License Plate Recognition (LPR)* component, which we added to the traffic monitoring scenario, can be connected to one or more *Cam* components. The implementation of our *LPR* components uses the JavaANPR library [36] to detect license plate numbers of observed vehicles. The recognized number combined with the timestamp and the location information received from the *Cam* component is then published as an event.

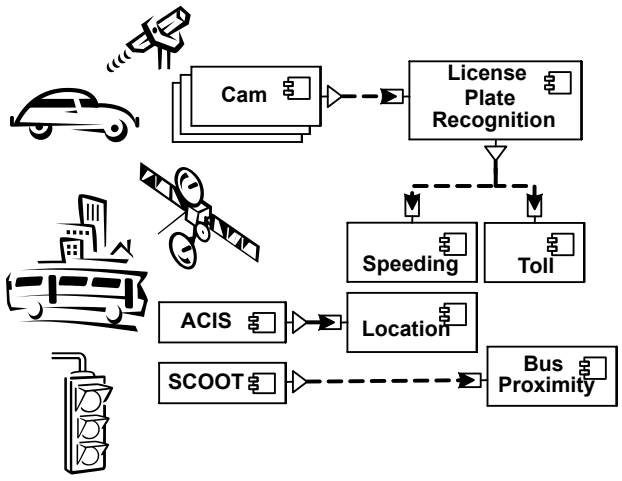


Fig. 16 Overview of Case Study Components

One component consuming the events of detected license plate numbers is the *Speeding* component. The component calculates the speed of a vehicle based on the distance between two cameras and the elapsed time between the two pictures.

Another component processing the events emitted by the *LPR* component is the *Toll* component. Assuming all arterial roads are equipped with *Cam* components, the *Toll* component determines the toll fee that must be payed for entering the city.

The *ACIS* component produces a stream of events, each containing a bus ID, a location, and the timestamp of the measurement. This data is collected by a set of sensors (in our case, GPS coupled with a proprietary radio network) to note the locations of buses and report them as they change.

The *Location Storage* component maintains a state that describes, for a set of objects, the most recent location that was reported for each of them. The input is a stream of events consisting of name/location pairs with timestamps, making *ACIS* a suitable event source.

In the city of Cambridge, the city's traffic lights are controlled by a SCOOT system [26], designed to schedule green and red lights to optimize the use of the road network. The *SCOOT* component is a wrapper of this system. It supplies a source endpoint emitting a stream of events corresponding to light status changes (red to green and green to red), a second source endpoint emitting a stream of events that reflects SCOOT's measurements of the traffic flow, and two RPC endpoints that allow retrieving information about junctions.

The *Bus Proximity* component receives a stream of events reflecting when lights turn from green to red. This stream is emitted by the *SCOOT* component. Upon such a trigger, the *SCOOT* component's RPC facility is used to determine the location of the light that just turned red. This is collated with current bus locations (stored in a relational database by the location storage component) to find which buses are nearby.

Performance Model The parametrization of PCM allows us to specify a *Repository* with reusable components that can be instantiated multiple times. This enables the modeling and evaluation of different system alternatives without changing the specifications of the components. To connect the components with the *Usage Model*, which specifies the rate of incoming events, we need some additional trigger interfaces. Thus, in addition to the event sinks and sources in Figure 16, the three components *ACIS*, *SCOOT*, and *Cam* provide such additional trigger interfaces. Except for the *LPR*, the resource demands of the components are nearly constant and independent

of the data values included in the event. This allows us to model them as fixed demands in an `InternalAction` of the respective RDSEFF. For each component, we measured the internal processing time under low system load and derived the resource demands. Measurements with different pictures showed, that the resource demands required by the *LPR* component highly depend on the content of the picture. PCM allows to specify parameter dependencies, however, it is not possible to quantify the content of a picture. Thus, we modeled the resource demand using a probability distribution. We systematically analyzed a set of 100 different pictures. For each image, we measured the mean processing time required by the recognition algorithm over 200 detection runs. The standard deviation was less then 2% of the mean value for all measurements. The measurements indicated that the processing of pictures that can be successfully recognized is nearly log-normal distributed ($\mu = 12.23, \sigma = 0.146$). Pictures where no license plate could be detected have a significantly higher but fixed processing time of 109.2ms. To represent this behavior in the RDSEFF of the *LPR* component, we used a `BranchAction`. One branch contains an `InternalAction` with the fixed demand for undetected images and the other one contains a log-normal distribution which we fitted to the measurements for successfully detected images.

The SBUS-specific *Middleware Repository* contains one component representing event sources and one representing event sinks. Both components include a semaphore to model the single threaded behavior of the SBUS implementation. Furthermore, the RDSEFFs include `InternalActions` to represent the resource demands required within the SBUS middleware. We instrumented the SBUS implementation to measure the processing time in the different event processing stages. In order to derive the CPU demand for each InternalAction, we extended the SBUS framework with several sensors that collect the time spent within a component itself, within the library to communicate with the wrapper, and within the wrapper to communicate with the library and the receiving component. For each component, we ran experiments and measured the time spent in the component, the library, and the wrapper under low workload conditions. We took the mean value over more than 10 000 measurements whose variation was negligible.

In the *System Model*, the components are instantiated and connected with each other depending on the design alternatives that should be analyzed.

According to the deployment option that should be analyzed, we use the *Allocation Model* to describe the allocation of components on individual hardware nodes. In our case study, the *ResourceEnvironment* describes our test environment, which consists of 8 `ResourceContainers`, each containing one `ProcessingResource` representing the CPU. We selected processor sharing on 4 cores as `SchedulingPolicy`, as all machines in our testbed are equipped with quad-core CPUs. The `ResourceContainers` are connected by a `LinkingResource` with a throughput of 1 GBit/s. The mapping of components to hardware nodes is adapted according to the individual deployment options in the scenarios.

The *Usage Model* consists of three different types of scenarios, which are executed in parallel. Two `UsageBehaviours` are used to trigger SCOOT and ACIS to emit events. For both behaviors, we specify an `OpenWorkload` with an exponentially distributed inter-arrival time with a mean value of 200ms. Additionally, we introduce a `UsageBehaviour` for each street equipped with two cameras. In these behaviors, the two calls of the cameras are separated by a `DelayAction`. With this equally distributed delay, we simulate the driving time of a vehicle from the first camera to the second one. Each *Cam* call includes the specification of the image size. Similar to the other behaviors, we use an exponentially distributed inter-arrival time for the first camera.

Evaluation Experiments To evaluate the prediction accuracy, we deployed the traffic monitoring application in different scenarios representing different evolutionary stages of the system and possible design alternatives in our testbed. We extended the implementations of *SCOOT*, *ACIS* and *Cam* with configurable and scalable event-generators. The events emitted by *SCOOT* and *ACIS* are based on an event stream recorded in the City of Cambridge. The event generator added to the *Cam* component uses a set of real pictures of different vehicles including their license plates. All event generators have in common that the event rate can be defined using a configuration file.

Our experimental environment (see Fig. 17) consisted of 12 identical machines, each equipped with a 2.4 GHz Intel Core 2 Quad Q6600 CPU, 8 GB main memory, and two 500 GB SATA II disks. All machines were running Ubuntu Linux version 8.04 and were connected through a GBit LAN. Our implementation of the components allow to replay a predefined event stream with a specified event-rate. This allows us to analyze the different deployment options under different load situations.

A single run of the prediction series simulates about 100000 pictures and its execution lasts about 3 minutes. On a real system, measuring such a set of data will last

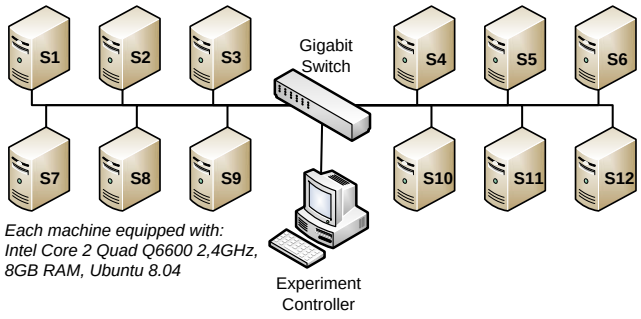


Fig. 17 Experiment Testbed

up to 5 hours and longer. For this reason, we had to limit the number of experiment runs and workload scenarios. For each scenario, we conducted up to seven experiments which cover the whole range from low to high load on the system. In the following, we describe the required changes to reflect the different design and deployment options in architecture-level models and present the results of these measurements compared to the predicted values.

Scenario 1: Deployment Variations In this scenario, three streets are equipped with cameras to monitor the traffic and two servers are available on which the system components can be deployed. The performance predictions are used to evaluate two different deployment options, namely all processing components on one system (AllOnOne Deployment) and *LPR* separated from the other processing components (Distributed Deployment). Thanks to the separation of the *Repository Model* and *System Model*, the instantiation and connections of the components can be done with little effort. Combined with the effort for specifying the component deployment and the *Usage Model*, the model was created in less than 30 minutes.

Figure 19(a) visualizes the predicted and measured mean CPU utilization of the machines hosting the *LPR* component as well as the machine hosting the remaining components in the distributed deployment. Overall, the mean prediction error of the CPU utilization in this scenario is less than 5%. In both deployment options, the prediction error increases with higher CPU load, which can be explained by caching effects since the algorithm used within the *LPR* component is very memory-intensive and the high CPU load leads to increasing number of context switches during execution. The measured utilization under the highest load in both options is lower than expected. The analysis of throughput measurements shows that images were queued up and not processed by the *LPR* component, if the CPU utilization is higher than 80%. This is an indicator for an overloaded and unstable system state. We conducted some more experiments running the system continuously over several hours as well as with an increased event rate. In both cases, the system crashed and completely halted. This confirms our assumption of an overloaded and unstable system state.

Scenario 2: New Components As all arterial roads in and out of the city centre are equipped with cameras, it is possible to monitor vehicles entering and leaving the inner city. This allows to build up an automated toll collection system, represented by the *Toll* component as a second component processing the events emitted by the *LPR* component. It induces additional load on the CPU, which was not foreseen in the previous scenarios. To increase the system's throughput, additional hardware is added and it is now possible to run three independent

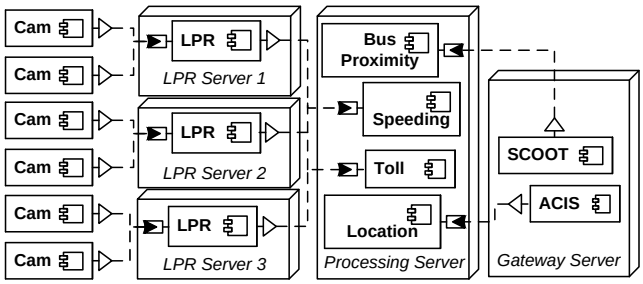


Fig. 18 Scenario 2: Centralized Deployment

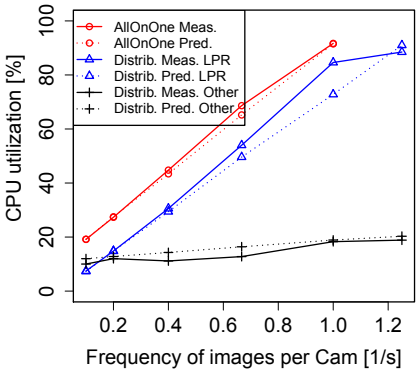
instances of *LPR* on different nodes. We again evaluated two deployment options. In both options, three individual instances of *LPR* are running on different nodes, each responsible for the events of two cameras. In the first case, all other components are running on one node (see Figure 18) (Centralized Deployment) and in the second case *Speeding* and *Toll* are deployed with three separate instances and co-located with the *LPR* instances on the three nodes (Decentralized Deployment). The models defined within the previous scenario can be reused and only small changes were required. These changes took less than 20 minutes.

Figure 19(b) shows the predicted and measured mean utilization of the machines that host the *LPR* component for both deployment options. Additionally, it includes the utilization of the machine hosting the processing components in the centralized deployment options. We leave out the values for the decentralized deployment options, as they are independent of the image frequency. Overall, the mean prediction error for the CPU utilization of the machine hosting the *LPR* component was 11.52% and never exceeded 20%. Additionally, we compared the measured and predicted processing time within the *LPR* component. The results are listed in Table 2 and visualized in Figure 19(d). Under the highest workload, the decentralized deployment option was overloaded and thus, no values are presented in the table and the figure. Due to the caching effects, which can not be predicted by the model, the prediction error increases with higher event rates respectively higher CPU utilization. However, the mean prediction error is still below 20%.

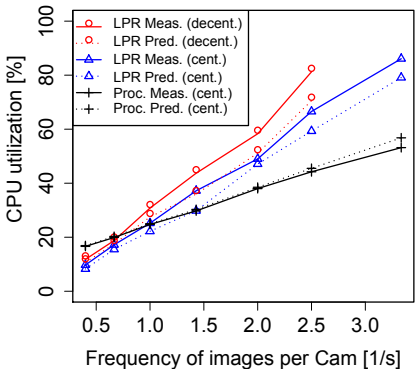
Scenario 3: Upgraded Hardware In this last scenario, the existing cameras can be replaced by a newer and improved version. The new cameras are able to take pictures with higher resolution and improved quality. With the improved quality, the detection error ratio can be reduced from 30% to 5%. It is known, that the resource demands for processing pictures with undetectable license plates are significantly higher than for successfully recognized license plates. However, the resource demands also depend on the image size. In this scenario, the impact of introducing the new camera version on the overall system performance is evaluated. This evaluation allows

Table 2 Scenario 2: LPR Mean Processing Time

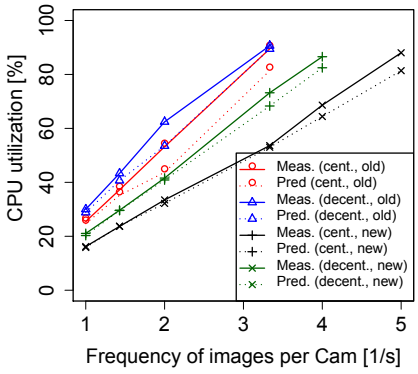
Image rate per Cam [1/s]:	0.4	0.67	1	1.43	2	2.5	3.33
Measurement(centralized) [s]:	0.47	0.48	0.49	0.55	0.66	0.84	1.99
Prediction (centralized) [s]:	0.41	0.47	0.44	0.43	0.52	0.59	0.96
Error (centralized) [%]:	12.4	2.0	10.0	21.7	21.7	30.4	52.1
Measurement (decentralized) [s]:	0.49	0.48	0.52	0.57	0.68	1.09	-
Prediction (decentralized) [s]:	0.44	0.47	0.44	0.44	0.49	0.73	-
Error (decentralized) [%]:	9.6	2.8	15.0	22.4	27.4	32.2	-



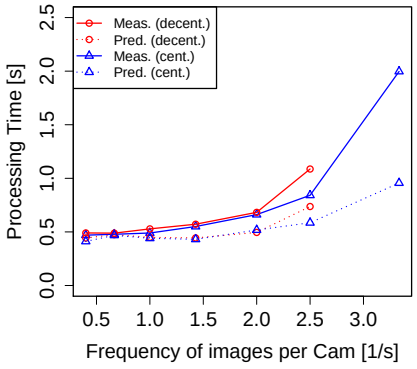
(a) Scenario 1: CPU Utilization



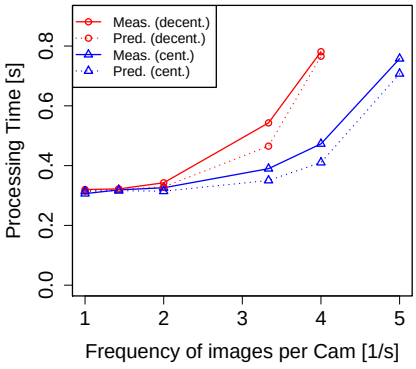
(b) Scenario 2: CPU Utilization



(c) Scenario 3: CPU Utilization



(d) Scenario 2: Processing Time



(e) Scenario 3 (New Camera): Processing Time

Fig. 19 Predicted and Measured Values of CPU Utilization and Processing Time

to decide if the investment into new cameras will improve the system performance. Similar to the previous scenario, we evaluate a centralized and a decentralized deployment of the *Toll* and *Speeding* components. To represent the new cameras in the prediction model, only two model parameters, the size of an image and the probability of an unsuccessful detection, must be changed. Additionally, the new *Cam* and *LPR* instances must be added to the composition and allocation models. Nevertheless, the required modeling time was less then 10 minutes. The results of the measurements and predictions of the mean CPU utilization of the machines hosting an instance of the *LPR* component are shown in Figure 19(c). Again, the prediction error increases with higher load due to the caching effects induced by the memory intensive algorithm of the *LPR*. However, the mean prediction error is only 5.56%. We also analyzed the measured and predicted mean processing time within the *LPR* component. In Figure 19(e), we present the processing times of *LPR* in the scenarios using the improved cameras. The mean prediction error is 5.36% and never exceeded 15%. Similar to Scenario 1, the measured CPU utilization and processing time in the decentralized deployment option are lower than expected as events are queued up again. The results for an even higher load which completely overloaded the system are not included.

Summary The different scenarios modeled and evaluated within the traffic monitoring case study, highlight the high adaptability of the model, enabling the easy evaluation of different design and deployment alternatives. As already mentioned in the different scenarios,

the required adaptation of the models can be done in less than 30 minutes in all cases. The modeling effort is negligible low compared to the specification of completely new performance models as required by many existing prediction techniques, and especially compared to setting up a test system to measure the effect of the considered changes. Furthermore, the execution of one simulation run, which consists of 100000 simulated events, takes about 3 minutes on a MacBook Pro with a Core i7 processor and 8 GB RAM. Assuming the highest event rate of five images per camera per second, this corresponds to a time span of 2.7 hours to collect the same amount of measurements in the testbed. Overall, the prediction error is less than 20% in most cases.

5.2.2 SPECjms2007 Benchmark The SPECjms2007 benchmark is based on a supply chain management scenario designed to be representative for real-world event-based applications. The benchmark was developed by SPEC's Java Subcommittee with the participation of IBM, Sun, Oracle, BEA Systems, Sybase, Apache, JBoss and TU Darmstadt. The benchmark workload comprises a set of supply chain interactions that represent a complex transaction mix exercising both point-to-point and publish/subscribe messaging including one-to-one, one-to-many and many-to-many communication [52]. The benchmark covers the major message types used in practice including messages of different sizes and different delivery modes, e.g., persistent vs. non-persistent and transactional vs. non-transactional. Due to its complexity and mix of interaction and workloads, SPECjms2007 is an ideal case study to demonstrate the applicability and expressiveness of our approach and allows us to evaluate the accuracy of the prediction results in complex and realistic scenarios with different workload mixes.

Scenario The application scenario is a supply chain management system of a supermarket company where RFID technology is used to track the flow of goods. The participants involved can be grouped into four roles:

1. *Supermarkets (SMs)* that sell goods to end customers,
2. *Distribution Centers (DCs)* that supply the supermarket stores,
3. *Suppliers (SPs)* that deliver goods to the distribution centers and
4. *Company Headquarters (HQ)* responsible for managing the accounting of the company.

SPECjms2007 implements seven interactions between the participants in the supply chain:

1. Order/shipment handling between SM and DC
2. Order/shipment handling between DC and SP
3. Price updates sent from HQ to SMs
4. Inventory management inside SMs
5. Sales statistics sent from SMs to HQ
6. New product announcements sent from HQ to SMs
7. Credit card hot lists sent from HQ to SMs

The workflow of the seven interactions is shown in Figure 20. Interactions 1, 4 and 5 exercise point-to-point messaging whereas interactions 3, 6 and 7 exercise publish/subscribe messaging. Interaction 2 contains both point-to-point and publish/subscribe messaging. A brief description of Interaction 2, which includes both point-to-point and publish/subscribe messaging, illustrates the complexity of the workload. The interaction is triggered when goods in a DC are depleted and the DC has to order from a SP to refill stock: i) a DC sends a call for offers to all SPs that supply the required types of goods, ii) SPs send offers to the DC, iii) the DC selects a SP and sends a purchase order to it, iv) the SP ships the ordered goods sending a confirmation and an invoice, v) the shipment is registered by RFID readers upon entering the DC's warehouse, vi) the DC sends a delivery confirmation to the SP, vii) the DC sends transaction statistics to the HQ.

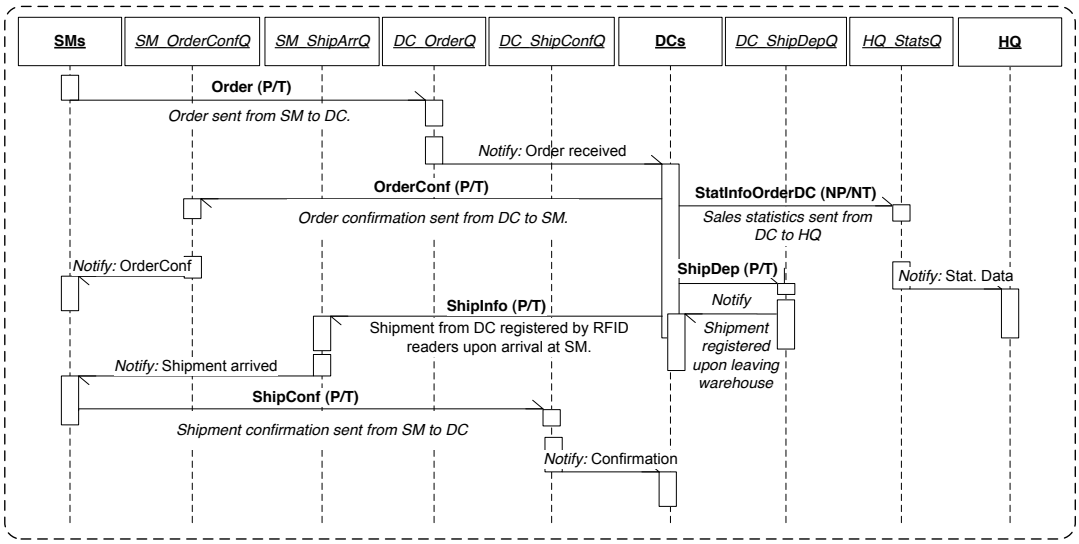
In this case study, we have intentionally slightly deviated from the standard system topology to avoid presenting performance results that may be compared against standard SPECjms2007 results. The latter is prohibited by the SPECjms2007 run and reporting rules. To this end, we use a topology based on the benchmark's vertical topology with 10 DC and HQ instances each set to 10. We studied the following workload scenarios:

- *Scenario 1:* A mix of all seven interactions exercising both point-to-point and publish/subscribe messaging.
- *Scenario 2:* A mix of interactions 4 and 5 focused on point-to-point messaging.
- *Scenario 3:* A mix of interactions 3, 6 and 7 focused on publish/subscribe messaging.

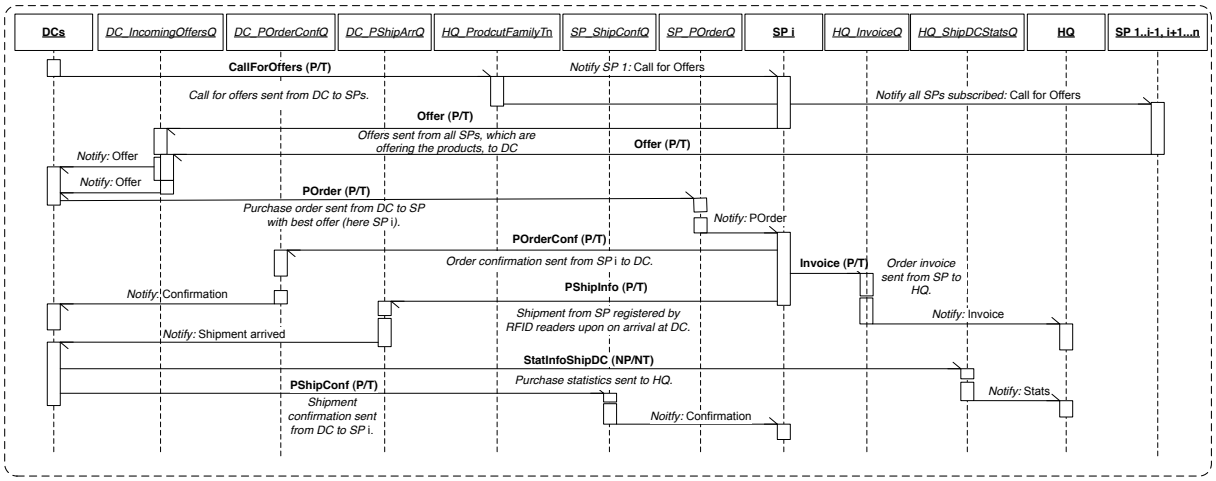
Table 3 provides a detailed workload characterization of the three scenarios to illustrate the differences in terms of transaction mix and message size distribution.

Prediction Model In the *Repository Model* each participant of the interactions is modeled as a separate component. Additionally, we specified the different event types that are used in the interactions and specified the emitted and accepted events for each component. The focus of SPECjms2007 is the evaluation of the underlying communication middleware. Thus, in contrast to the traffic monitoring case study, the business logic of the different component implementations is simplified to reduce the influences of the component implementations on the overall system performance. Similar to the traffic monitoring case study, we added interfaces to trigger the interaction.

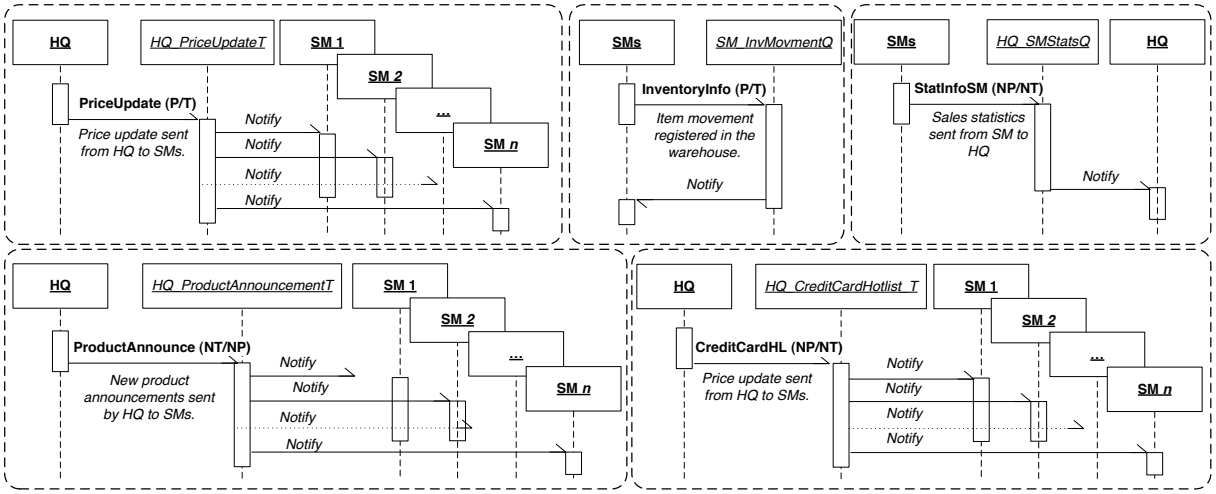
As the SPECjms2007 benchmark focuses on evaluating the performance of the message-oriented middleware, the specification and calibration of the *Middleware Repository* is an important factor for the accuracy of the prediction results. For each event type, we identified the resource demands for the CPU, HDD and



(a) Interaction 1



(b) Interaction 2



(c) Interactions 3 to 7

Fig. 20 Workflow of the SPECjms2007 Interactions: (N)P=(Non-)Persistent, (N)T=(Non-)Transactional

Table 3 Scenario Transaction Mix

	Sc. 1			*Sc. 2*	*Sc. 3*
	In	*Out*	*All*		
No. of Msg.					
P2P					
- P/T	49.2%	40.7%	44.6%	21.0%	-
- NP/NT	47.2%	39.0%	42.8%	79.0%	-
Pub/Sub					
- PT	1.8%	6.0%	4.1%	-	17.0%
- NP/NT	1.7%	14.2%	8.5%	-	83.0%
Overall					
- PT	51.1%	46.7%	48.7%	21.0%	17.0%
- NT/NP	48.9%	53.3%	51.3%	79.0%	83.0%
Traffic					
P2P					
- P/T	32.2%	29.5%	30.8%	11.0%	-
- NP/NT	66.6%	61.0%	63.5%	89.0%	-
Pub/Sub					
- PT	0.5%	2.3%	1.6%	-	3.0%
- NP/NT	0.8%	7.2%	4.1%	-	97.0%
Overall					
- PT	32.7%	31.8%	32.4%	11.0%	3.0%
- NT/NP	67.3%	68.2%	67.6%	89.0%	97.0%
Avg. Size	*(in KBytes)*				
P2P					
- P/T	2.13			2.31	-
- NP/NT	4.59			5.27	-
Pub/Sub					
- PT	1.11			-	0.24
- NP/NT	1.49			-	1.49
Overall					
- PT	2.00			2.31	0.24
- NT/NP	3.76			5.27	1.49

For Scenario 2 &3: In = Out.

LAN resources. We estimated the demands by running the interactions in isolation and measuring the utilization of the respective resources using OS tools on the sender, middleware and sink sides. For interactions consisting of multiple messages, the demands of the individual messages were estimated by considering their relative fraction of the whole interaction. To derive the demands of notification messages, we repeated the experiments with different numbers of subscribers and used linear regression to estimate the service demands. To reflect the resource demands on the source side, we specified the component `JMSSource` providing the interface `handleSourcePort` with an RDSEFF containing the event type dependent resource demands modeled within a `GuardedBranchAction`. Similarly, we defined a `JMSSink` component, reflecting the event type specific resource demands induced on the receiver side. In the `JMSServer` component, which reflects the middleware, we need to distinguish between the resource demands induced by a message received from sources and the messages sent to all subscribed sinks. Especially in case of publish/subscribe communication, this separation is essential, as the

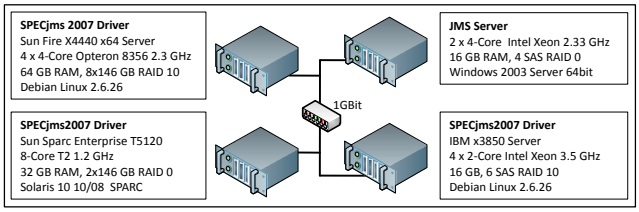


Fig. 21 Experimental Environment

resource demands for forwarding messages to the subscribed sinks depend on the number of subscribed sinks. For this reason, we specified two RDSEFFs within the JMSServer component. One implements the `handleDistributionPreparation` interface. It includes the event type and size dependent resource demands on the CPU, HDD and LAN required for processing the message received from a source. The other RDSEFF implements the `handleSender` interface and contains the resource demands required for delivering the message to one of the subscribed sinks.

Corresponding to the SPECjms2007 system topology, we instantiated each component (SM, DC, HQ, and SP) several times within the *System Model*. According to the different interactions (see Figure 20), we connected the different component instances. In case of point-to-point communication, we used the direct point-to-point connector between sources and sinks. In case of publish/subscribe communication we first defined the respective `EventChannel` and than connected sources and sinks with this channel.

The *Resource Environment* consists of several `ResourceContainers`. We defined the available resources according to the hardware available in our testbed (see Figure 21). For example, the `ResourceContainer` hosting the middleware includes a `ProcessingResource` with processor sharing scheduling on 8 cores to model the CPU and two first-come-first-serve resources for the LAN and HDD. In the allocation model, we deployed the different components on the corresponding `ResourceContainer`. The deployment of the middleware components on the middleware container is automatically done by the transformation described in Section 4.5.

In the *Usage Model*, we specified a dedicated `UsageProfile` for each interaction. Within each of these `UsageProfiles`, we added a call to the trigger interfaces we specified within the *Repository Model*. Using separate `UsageProfiles` enables us to specify individual rates for each interaction or completely deactivate them if necessary.

Evaluation Experiments To evaluate the accuracy of our modeling approach, we conducted an experimental analysis of the modeled application in the environment depicted in Figure 21. A leading commercial message-oriented middleware platform was used as a JMS server installed on a machine with two quad-core Intel Xeon

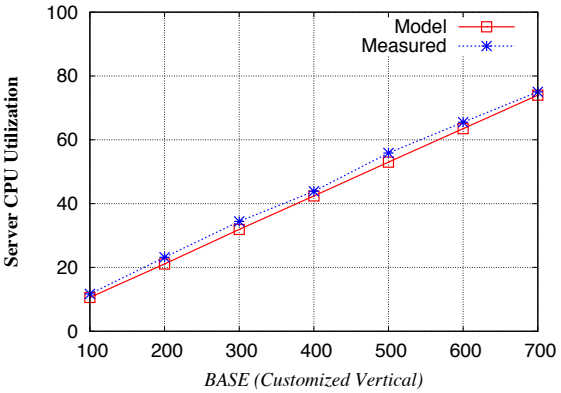


Fig. 22 Server CPU Utilization for Customized Vertical Topology

2.33 GHz CPUs and 16 GB of main memory. The server was operated in a 64-bit 1.5 JVM with 8GB of heap space. A RAID 0 disk array comprised of four disk drives was used for maximum performance. The JMS Server was configured to use a file-based store for persistent messages with a 3.8 GB message buffer. The workload drivers were distributed across three machines: i) one Sun Fire X4440 x64 server with four quad-core Opteron 2.3 GHz CPUs and 64 GB of main memory, ii) one Sun Sparc Enterprise T5120 server with one 8-core T2 1.2 GHz CPU and 32 GB of main memory and iii) one IBM x3850 server with four dual-core Intel Xeon 3.5 GHz CPUs and 16 GB of main memory. All machines were connected to a 1 GBit network.

In each case, the model was analyzed using simulation with at least 100000 simulated transactions in each simulation run. The SPECjms2007 benchmark provides a central parameter named *BASE* to configure the induced workloads. Figure 22 shows the predicted and measured CPU utilization of the message-oriented middleware server for the considered customized vertical topology when varying the *BASE* between 100 and 700. As we can see, the model predicts the server CPU utilization very accurately as the workload is scaled. In the following, we present a more detailed evaluation of the three scenarios under different load intensities considering further performance metrics such as interaction throughput and completion time.

The detailed results for the scenarios are presented in Table 4 and illustrated in Figure 23. For each scenario, we consider two workload intensities corresponding to medium and high load conditions configured using the *BASE* parameter. The first scenario represents the vertical interaction mix for *BASE* 300 and 550, respectively. The second scenario is a mix of interaction 4 and 5 focused on point-to-point communication, while the third scenario is a mix of interaction 3, 6 and 7 focused on publish/subscribe communication. For each scenario, the interaction rates and the average interaction completion times are shown. The *interaction completion time* is defined as the time between the beginning of the interaction and the time when the last message has been processed. The difference between the predicted and measured interaction rates was negligible (with an error below 1%) and therefore we only show the predicted interaction rates. For completion times, we show both, the predicted and measured mean values, where for the latter we provide a 95% confidence interval from 5 repetitions of each experiment. Given that the measured mean values were computed from a large number of observations, their respective confidence intervals were quite narrow. The prediction error was less than 25% in most cases. Especially in the cases where the interaction completion times are below 3 ms, e.g., for interaction 6 and 7 in the first scenario, the prediction error was higher. In such cases, a small absolute difference of 1 ms between the measured and predicted values (e.g., due to some synchronization aspects not captured by the model) appears high when considered as a percentage of

Table 4 Detailed Results for Scenario 1,2 and 3

(a) Scenario 1

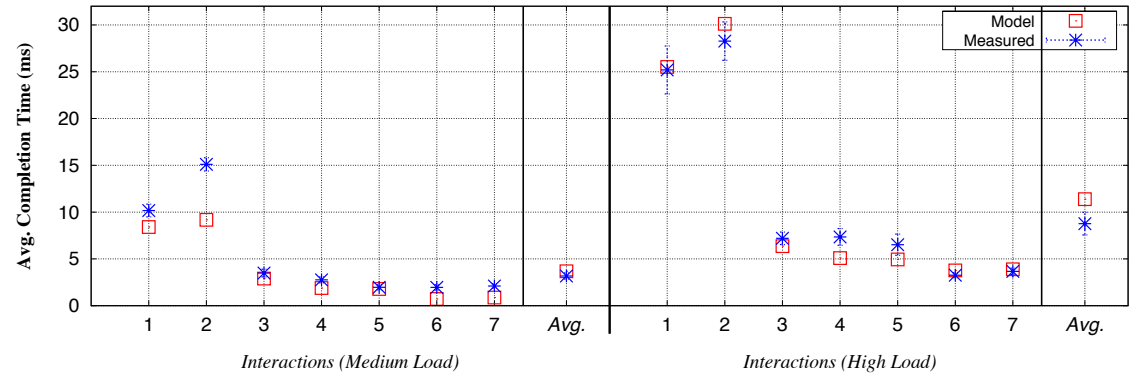
Input BASE	*Inter-action*	*Rate p. sec*	*Avg. Completion T (ms)* Model	Meas. (95% c.i.)
300 *med. load*	1	226.36	8.41	10.17 +/- 0.68
	2	66.9	9.18	15.10 +/- 0.71
	3	14.92	2.9	3.49 +/- 0.41
	4	483.4	1.89	2.76 +/- 0.31
	5	1734.7	1.79	1.97 +/- 0.27
	6	43.45	0.72	1.96 +/- 0.29
	7	30.65	0.87	2.10 +/- 0.24
550 *high load*	1	418.1	25.51	25.19 +/- 2.56
	2	120.15	30.12	28.27 +/- 2.05
	3	26.0	6.36	7.20 +/- 0.67
	4	887.5	5.09	7.35 +/- 0.89
	5	3189.4	4.94	6.52 +/- 1.13
	6	81.73	3.77	3.26 +/- 0.26
	7	56.9	3.89	3.67 +/- 0.34

(b) Scenario 2

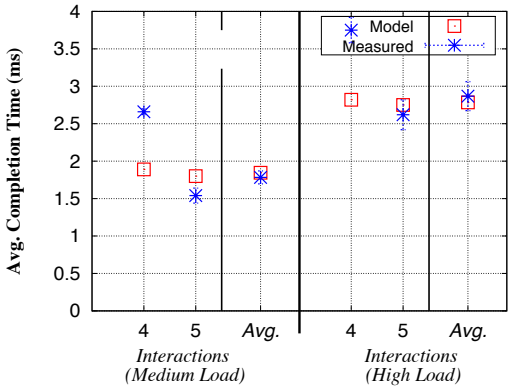
Input BASE	*Inter-action*	*Rate p. sec*	*Avg. Completion T (ms)* Model	Meas. (95% c.i.)
600 *med. load*	4	977.8	1.89	2.66 +/- 0.04
	5	3474.8	1.80	1.54 +/- 0.10
800 *high load*	4	1289.1	2.82	3.75 +/- 0.17
	5	4637.62	2.75	2.62 +/- 0.20

(c) Scenario 3

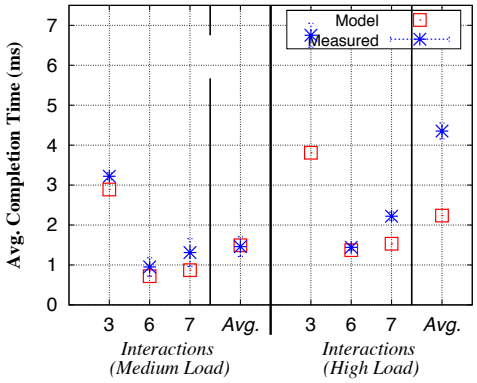
Input BASE	*Inter-action*	*Rate p. sec*	*Avg. Completion T (ms)* Model	Meas. (95% c.i.)
6000 *med. load*	3	304.1	2.89	3.22 +/- 0.09
	6	852.2	0.72	0.95 +/- 0.23
	7	617.9	0.87	1.31 +/- 0.35
10000 *high load*	3	498.3	3.81	6.75 +/- 0.30
	6	1418.2	1.37	1.44 +/- 0.07
	7	1025.53	1.53	2.22 +/- 0.10



(a) Scenario 1



(b) Scenario 2



(c) Scenario 3

Fig. 23 Predicted and Measured Completion Time

the respective mean value given that the latter is very low. However, when considered as an absolute value, the error is still quite small.

Figure 23 depicts the predicted and measured interaction completion times for the three scenarios. The results reveal the accuracy of the model when considering different types of messaging. For point-to-point messaging, the modeling error is independent of whether persistent or non-persistent messages are sent. However, for the publish/subscribe case under high load (Scenario 3), the modeling error is much higher for the case of persistent messages than for the case of non-persistent messages. In scenario 1 where all interactions are running at the same time, interaction 1 and 2 exhibited the highest modeling error (with exception of the interactions with very low completion times). This is due to the fact that each of these interactions comprise a complex chain of multiple messages of different types and sizes. Finally, looking at the mean completion time over all interactions, we see that the prediction is optimistic as the predicted completion times are lower than the measured ones. This behavior is typical for performance models in general since no matter how representative they are, they normally cannot capture all factors causing delays in the system.

Summary In summary, the model proved to be very accurate in predicting the system performance, especially considering the size and complexity of the system that was modeled. The prediction error does not exceed 20% in most cases. As discussed above, in cases where interaction completion times were below 3 ms, the relative prediction error was higher. Nevertheless, the absolute prediction error was less than 2ms. The case study demonstrates that the proposed modeling and prediction approach is applicable for complex and realistic industrial systems. In contrast to the traffic monitoring case study, the SPECjms2007 benchmark allows us to evaluate the modeling and prediction of point-to-point as well as publish/subscribe communication and even provides scenarios with mixed workloads.

5.2.3 Overall Evaluation Summary Architecture level performance models as presented, enable the evaluation of different design and deployment options at design time with very low effort as the adaptation of the models can be done supported by the graphical editors in few minutes. With the presented extensions combined with the automated transformation, the modeling effort to reflect event-based communication in the performance model could be reduced significantly by more

than 80% compared to the original PCM not supporting the explicit modeling of event-based communication. We demonstrated the applicability of our approach supporting the evaluation of different design and deployment questions typically for event-based systems. Additionally, we demonstrated the ability of our approach to handle complex systems with different communication styles and workload mixes within the SPECjms2007 case study. In both case studies, the prediction accuracy was less then 20% in most cases. With the aim of evaluating and comparing different design and deployment alternatives at design time, the accuracy is more than acceptable [39]. At system deployment and run-time, the prediction models help to detect system bottlenecks and to ensure that sufficient resources are allocated to meet performance and QoS requirements. Considering that nowadays systems are normally over-provisioned by a factor of two or more [28], the accuracy of our prediction results are sufficient to improve the capacity planning and run-time management of event-based systems.

6 Related Work

The work related to the results presented in this paper can be classified into two areas: i) architecture-level performance meta-models for component-based systems and ii) performance analysis techniques specialized for event-based systems.

Following the SPE [55] approach, a number of architecture-level performance meta-models have been developed. Often, they are based on the UML as the de facto standard modeling language for software architectures like the UML SPT profile [44] and its successor the UML MARTE profile [45]. Architecture-level performance models are built during system development and are used at design and deployment time to evaluate alternative system designs and/or predict the system performance for capacity planning purposes.

In recent years, with the increasing adoption of component-based software engineering, the performance evaluation community has focused on adapting and extending conventional SPE techniques to support component-based systems, which are typically used for building modern software systems. A recent survey of methods for component-based performance-engineering was published in [33].

Several approaches use model transformations to derive performance prediction models (e.g., [37,46,13,6]). Cortellessa et al. surveyed three performance meta-models in [10] leading to a conceptual MDA framework of different model transformations for the prediction of different extra-functional properties [12,11]. The influence of certain architectural patterns on the system performance and their integration into prediction models was studied by Petriu [46,19] and Gomaa [17].

Petriu et al. [46,19] modeled the pipe-and-filter and client-server architectural patterns using the UML. The models are transformed into LQNs using graph transformations as well as XSLT transformations. Gomaa and Menasce [17] developed performance models for component interconnections in client/server systems based on typical interconnection patterns.

Happe et al. [20] present a method for modeling message-oriented middleware systems using performance completions. Model-to-model transformations are used to integrate low-level details of the message-oriented middleware system into high-level software architecture models. A case study based on parts of the SPECjms2007 workload is presented as a validation of the approach. However, this approach only allows to model point-to-point connections using JMS queues.

In [57], Verdickt et al. present a framework to automatically include the impact of the CORBA middleware on the performance of distributed systems. Transformations map high-level middleware-independent UML models to UML models with middleware-specific information. The work focuses on the influence of Remote Procedure Calls (RPCs) as implemented in CORBA, Java RMI, and SOAP. The influence of service parameters on the performance was not considered. For example, the prediction model for marshaling and de-marshaling of service calls in [57] neglects the influence of the service's parameters.

The Platform-independent Component Modeling Language (PICML)[4] is part of the Component Synthesis with Model Integrated Computing (CoSMIC) modeling tool [16], developed at the Vanderbilt University. PICML provides a language to describe components in a platform-independent way. Different automated transformations are used to generate platform-specific code skeletons, deployment descriptors, and configuration files. As the main goal of CoSMIC and PICML is the generation of implementation artifacts, it lacks several informations required for performance predictions, like an explicit usage model of the system or a description of the component internal behavior. Furthermore, PICML only supports direct connections between components and does not provide meta-model elements to model event channels or a central event bus.

In the following, we present an overview of existing performance modeling and analysis techniques specialized for event-based systems. A survey of techniques for benchmarking and performance modeling of event-based systems was published in [31].

In [35], an approach to predicting the performance of messaging applications based on Java EE is proposed. The prediction is carried out during application design, without access to the application implementation. This is achieved by modeling the interactions among messaging components using queueing network models, calibrating the performance models with architecture attributes, and populating the model parameters using a lightweight application-independent benchmark. How-

ever, again the workloads considered do not include multiple message exchanges or interaction mixes.

Sachs et al. [53, 51] present a modeling approach based on an extend QPN formalism simplifying the modeling of software resources like queues and event channels. A set of modeling patterns supports the architect in specifying a QPN-based performance models. These patterns map architecture-level elements (e.g., event queues, publish/subscribe communication, or thread pools) to QPN submodels, that are later composed to built the system's performance model. The approach is demonstrated and validated using the SPECjms2007 benchmark as representative case study.

In [22], an analytical model of the message processing time and throughput of the WebSphereMQ JMS server is presented and validated through measurements. The message throughput in the presence of filters is studied and it is shown that the message replication grade and the number of installed filters have a significant impact on the server throughput. Several similar studies using Sun Java System MQ, FioranoMQ, ActiveMQ, and BEA WebLogic JMS server were published. A more in-depth analysis of the message waiting time for the FioranoMQ JMS server is presented in [40]. The authors study the message waiting time based on an $M/G/1-\infty$ queue approximation and perform a sensitivity analysis with respect to the variability of the message replication grade. They derive formulas for the first two moments of the message waiting time based on different distributions (deterministic, Bernoulli and binomial) of the replication grade. These publications, however, only consider the overall message throughput and latency and do not provide any means to model event-based communication and message flows.

Several performance modeling techniques specifically targeted at distributed publish/subscribe systems exist in the literature. However, these techniques are normally focused on modeling the routing of events through distributed broker topologies from publishers to subscribers as opposed to modeling interactions and message flows between communicating components in event-based systems. In [42] an analytical model of publish/subscribe systems that use hierarchical identity-based routing is presented. The model is based on continuous time birth-death Markov chains. This work, however, only considers routing table sizes and message rates as metrics and the proposed approach suffers from several restrictive assumptions limiting its practical applicability. In [32, 31], a methodology for workload characterization and performance modeling of distributed event-based systems is presented. A workload model of a generic system is developed and analytical analysis techniques are used to characterize the system traffic and to estimate the mean notification delivery latency. For more accurate performance prediction queueing Petri net models are used. While the results are promising, the technique relies on monitoring data obtained from the system during operation which limits its applicability for design-time predictions.

7 Conclusion and Outlook

In this paper we presented i) a modeling approach for event-based communication at the architecture level exemplarily implemented based on PCM, ii) a two-step transformation approach enabling the performance prediction of the system including the consideration of platform-specific middleware influence factors, and iii) a detailed evaluation of the presented approach based on two real-world case studies representing different domains of event-based systems.

The presented meta-model elements allow architects to model event-based systems at the architecture levels. Introducing events as first class entity enables the architect to specify individual source and sink ports for components. The presented approach enables to differentiate between direct point-to-point and decoupled publish/subscribe communication using dedicated event channels.

The developed two-step transformation refines the event-based connections between components. The transformation is partitioned into a platform-independent and a platform-specific part. In the first part, the new elements are transformed to a set of elements, following a generic event processing chain. In the second step, platform-specific components located in a separate middleware repository are woven into the prediction model. Due to this separation, the influence of using different middleware systems can be analyzed by simply selecting another middleware repository and the system itself can be modeled independent of the underlying middleware. Furthermore, the transformation allows a semantically correct modeling of event-based communication using the introduced meta-model elements while still supporting all existing prediction techniques such as simulation [6], LQNs [34] or QPNs [38].

We evaluated our approach based on two representative real-world case studies: A distributed traffic monitoring system built on top of the peer-to-peer middleware SBUS and the SPECjms2007 benchmark, a representative supply chain system using a centralized middleware supporting the JMS standard with complex and varying workload mixes. The results show, that using the presented meta-model elements the modeling effort is reduced by more than 80%. Applying our approach to different design and deployment alternatives of the traffic monitoring case study, allows us to demonstrated the adaptability of the models and applicability of our approach to support an architect in evaluating different design decisions. The prediction error was less than 20% in most cases for both case studies. This demonstrates that the presented modeling and prediction approach can be applied at design time, to evaluate and compare different design alternatives, as well as at deployment time, to

analyze different deployment options and to determine the required hardware resources.

The results presented in this paper form the basis for several areas of future work. In the presented metamodel elements, filtering of events has to be modeled manually in the behavior model of the sink. The presented approach requires the existence of a platform-specific middleware repository. The Performance Cockpit approach [58] uses automated experiments to derive parameterized resource demands for components. As a next step, we plan to define a set of experiments an automated generation of the middleware repository model using the Performance Cockpit. Furthermore, our current and future research focuses on the idea of making architecture-level performance models usable at runtime. The Descartes Research Group [1] is working on enhancing design-time models to capture dynamic aspects of the environment and making them an integral part of the system [30]. To achieve this, the execution environment should be enhanced with functionality to track dynamic changes and automatically maintain the prediction models during operation. The initial models can either be built manually during system design as presented in this paper or they can be extracted at runtime based on online monitoring and measurement data as advocated in [7].

References

1. Descartes Research Group ; http://www.descartes-research.net.
2. Palladio simulator; http://www.palladio-simulator.com.
3. J. Bacon, A. R. Beresford, D. Evans, D. Ingram, N. Trigoni, A. Guitton, and A. Skordylis. TIME: An open platform for capturing, processing and delivering transport-related data. In *Proceedings of the IEEE consumer communications and networking conference*, pages 687–691, 2008.
4. K. Balasubramanian, J. Balasubramanian, J. Parsons, A. Gokhale, and D. C. Schmidt. A platform-independent component modeling language for distributed real-time and embedded systems. *J. Comput. Syst. Sci.*, 73(2):171–185, Mar. 2007.
5. S. Becker. *Coupled Model Transformations for QoS Enabled Component-Based Software Design*, volume 1 of *Karlsruhe Series on Software Design and Quality*. Universitätsverlag Karlsruhe, 2008.
6. S. Becker, H. Koziolek, and R. Reussner. The Palladio component model for model-driven performance prediction. *Journal of Systems and Software*, 82:3–22, 2009.
7. F. Brosig, N. Huber, and S. Kounev. Automated Extraction of Architecture-Level Performance Models of Distributed Component-Based Systems. In *26th IEEE/ACM International Conference On Automated Software Engineering (ASE 2011)*, Oread, Lawrence, Kansas, November 2011. To appear. Acceptance Rate (Full Paper): 14.7% (37/252).
8. A. Carzaniga, E. Di Nitto, D. S. Rosenblum, and A. L. Wolf. Issues in supporting event-based architectural styles. In *Proceedings of the third international workshop on Software architecture*, ISAW '98, pages 17–20, New York, NY, USA, 1998. ACM.
9. S. Castelli, P. Costa, and G. P. Picco. Modeling the communication costs of content-based routing: the case of subscription forwarding. In *DEBS '07: Proceedings of the international conference on Distributed Event-based Systems*, pages 38–49, New York, NY, USA, 2007. ACM.
10. V. Cortellessa. How far are we from the definition of a common software performance ontology? In *WOSP '05: Proceedings of the 5th International Workshop on Software and Performance*, pages 195–204, New York, NY, USA, 2005. ACM Press.
11. V. Cortellessa, A. Di Marco, and P. Inverardi. Integrating Performance and Reliability Analysis in a Non-Functional MDA Framework. In M. B. Dwyer and A. Lopes, editors, *Fundamental Approaches to Software Engineering, 10th International Conference, FASE 2007, Held as Part of the Joint European Conferences, on Theory and Practice of Software, ETAPS 2007, Braga, Portugal, March 24 - April 1, 2007, Proceedings*, volume 4422 of *Lecture Notes in Computer Science*, pages 57–71. Springer, 2007.
12. V. Cortellessa, P. Pierini, and D. Rossi. Integrating Software Models and Platform Models for Performance Analysis. *IEEE Transactions on Software Engineering*, 33(6):385–401, June 2007.
13. A. Di Marco and P. Inveradi. Compositional Generation of Software Architecture Performance QN Models. In *Proceedings of WICSA 2004*, pages 37–46, 2004.
14. P. T. Eugster, P. A. Felber, R. Guerraoui, and A.-M. Kermarrec. The many faces of publish/subscribe. *ACM Computing Surveys*, 35:114–131, June 2003.
15. D. Evans, J. Bacon, A. R. Beresford, R. Gibbens, and D. Ingram. Time for change. In *Intertraffic World, Annual Showcase*, pages 52–56, 2010.
16. A. Gokhale, B. Natarjan, D. C. Schmidt, A. Nechypurenko, N. Wang, J. Gray, S. Neema, T. Bapty, and J. Parsons. Cosmic: An mda generative tool for distributed real-time and embdedded component middleware and applications. In *Proceedings of the OOPSLA 2002 Workshop on Generative Techniques in the Context of Model Driven Architecture*, Seattle, WA, 2002.
17. H. Gomaa and D. A. Menascé. Design and performance modeling of component interconnection patterns for distributed software architectures. In *Proceedings of the second international workshop on Software and performance*, pages 117–126. ACM Press, 2000.
18. V. Grassi, R. Mirandola, and A. Sabetta. From Design to Analysis Models: a Kernel Language for Performance and Reliability Analysis of Component-based Systems. In *WOSP '05: Proceedings of the 5th international workshop on Software and performance*, pages 25–36, New York, NY, USA, 2005. ACM Press.
19. G. P. Gu and D. C. Petriu. Xslt transformation from uml models to lqn performance models. In *Proceedings of the third international workshop on Software and performance*, pages 227–234. ACM Press, 2002.
20. J. Happe, S. Becker, C. Rathfelder, H. Friedrich, and R. H. Reussner. Parametric Performance Completions for Model-Driven Performance Prediction. *Performance Evaluation*, 67(8):694–716, 2010.

21. J. Happe, H. Koziolek, and R. Reussner. Facilitating performance predictions using software components. *IEEE Software*, 28(3):27 –33, may-june 2011.
22. R. Henjes, M. Menth, and C. Zepfel. Throughput Performance of Java Messaging Services Using WebsphereMQ. In *Distributed Computing Systems Workshops, 2006. ICDCS Workshops 2006. 26th IEEE International Conference on*, 2006.
23. A. Hinze and A. P. Buchmann, editors. *Principles and Applications of Distributed Event-Based Systems*. IGI Global, 2010.
24. A. Hinze, K. Sachs, and A. Buchmann. Event-based applications and enabling technologies. In *Proceedings of the Third ACM International Conference on Distributed Event-Based Systems*, DEBS '09, pages 1:1–1:15, New York, NY, USA, 2009. ACM.
25. G. Hohpe and B. Woolf. *Enterprise integration patterns*. Addison-Wesley, 2008.
26. P. B. Hunt, D. I. Robertson, R. D. Bretherton, and R. I. Winton. SCOOT—a traffic responsive method of coordinating signals. Technical Report LR1014, Transport and Road Research Laboratory, 1981.
27. D. Ingram. Reconfigurable middleware for high availability sensor systems. In *Proceedings of the Third ACM International Conference on Distributed Event-Based Systems*, DEBS '09, pages 20:1–20:11, New York, NY, USA, 2009. ACM.
28. J. M. Kaplan, W. Forrest, and N. Kindler. Revolutionizing data center energy efficiency. Technical report, McKinsey&Company, 2008.
29. B. Klatt, C. Rathfelder, and S. Kounev. Integration of Event-Based Communication in the Palladio Software Quality Prediction Framework. In *7th ACM SIGSOFT International Conference on the Quality of Software Architectures (QoSA 2011)*, Boulder, Colorado, USA, June 20-24 2011.
30. S. Kounev. Engineering of Next Generation Self-Aware Software Systems: A Research Roadmap. In *Emerging Research Directions in Computer Science. Contributions from the Young Informatics Faculty in Karlsruhe*. KIT Scientific Publishing, July 2010. ISBN: 978-3-86644-508-6.
31. S. Kounev and K. Sachs. Benchmarking and Performance Modeling of Event-Based Systems. *it - Information Technology*, 5, Sept. 2009.
32. S. Kounev, K. Sachs, J. Bacon, and A. Buchmann. A methodology for performance modeling of distributed event-based systems. In *Proc. of the 11th IEEE Intl. Symposium on Object/Component/Service-oriented Real-time Distributed Computing*, May 2008.
33. H. Koziolek. Performance evaluation of component-based software systems: A survey. *Elsevier Performance Evaluation*, 67(8):634–658, August 2010.
34. H. Koziolek and R. Reussner. A Model Transformation from the Palladio Component Model to Layered Queueing Networks. In *Performance Evaluation: Metrics, Models and Benchmarks, SIPEW 2008*, volume 5119 of *Lecture Notes in Computer Science*, pages 58–78. Springer-Verlag Berlin Heidelberg, 2008.
35. Y. Liu and I. Gorton. Performance Prediction of J2EE Applications Using Messaging Protocols. *Component-Based Software Engineering*, pages 1–16, 2005.
36. O. Martinsky. Javaanpr - automatic number plate recognition system. `http://javaanpr.sourceforge.net/`, 2006.
37. M. Marzolla. *Simulation-Based Performance Modeling of UML Software Architectures*. PhD Thesis TD-2004-1, Dipartimento di Informatica, Università Ca' Foscari di Venezia, Mestre, Italy, Feb. 2004.
38. P. Meier, S. Kounev, and H. Koziolek. Automated Transformation of Palladio Component Models to Queueing Petri Nets. In *In 19th IEEE/ACM International Symposium on Modeling, Analysis and Simulation of Computer and Telecommunication Systems (MASCOTS 2011)*, Singapore, July 25-27 2011.
39. D. A. Menascé, V. A. F. Almeida, and L. W. Dowdy. *Performance by Design*. Prentice Hall, 2004.
40. M. Menth and R. Henjes. Analysis of the Message Waiting Time for the FioranoMQ JMS Server. In *Proc. of ICDCS '06*, Washington, DC, USA, 2006.
41. G. Mühl, L. Fiege, and P. R. Pietzuch. *Distributed Event-Based Systems*. Springer, 2006.
42. G. Mühl, A. Schröter, H. Parzyjegla, S. Kounev, and J. Richling. Stochastic Analysis of Hierarchical Publish/Subscribe Systems. In *Proceedings of the 15th International European Conference on Parallel and Distributed Computing (Euro-Par 2009), Delft, The Netherlands, August 25-28, 2009*. Springer Verlag, 2009.
43. O. OMG. Mda guide v1.0.1. `http://www.omg.org/cgi-bin/doc?omg/03-06-01`, 2003.
44. O. OMG. UML Profile for Schedulability, Performance, and Time (SPT), v1.1, Jan. 2005.
45. O. OMG. UML Profile for Modeling and Analysis of Real-Time and Embedded systems (MARTE), May 2006.
46. D. C. Petriu and X. Wang. From UML description of high-level software architecture to LQN performance models. In M. Nagl, A. Schürr, and M. Münch, editors, *Proc. of AGTIVE'99 Kerkrade*, volume 1779. Springer, 2000.
47. C. Rathfelder. *Modelling Event-Based Interactions in Component-Based Architectures for Quantitative System Evaluation*. PhD thesis, Karlsruhe Institute of Technology (KIT), Dezember 2012.
48. C. Rathfelder, D. Evans, and S. Kounev. Predictive Modelling of Peer-to-Peer Event-driven Communication in Component-based Systems. In A. Aldini, M. Bernardo, L. Bononi, and V. Cortellessa, editors, *Proceedings of the 7th European Performance Engineering Workshop (EPEW'10), University Residential Center of Bertinoro, Italy*, volume 6342 of *Lecture Notes in Computer Science*, pages 219–235. Springer-Verlag Berlin Heidelberg, 2010.
49. C. Rathfelder, B. Klatt, S. Kounev, and D. Evans. Towards Middleware-aware Integration of Event-based Communication into the Palladio Component Model (Poster Paper). In *Proceedings of the 4th ACM International Conference on Distributed Event-Based Systems (DEBS-2010)*, Cambridge, United Kingdom, July 2010. ACM, New York, USA.
50. C. Rathfelder and S. Kounev. Modeling Event-Driven Service-Oriented Systems using the Palladio Component Model. In *Proceedings of the 1st International Workshop on the Quality of Service-Oriented Software Systems (QUASOSS)*, pages 33–38. ACM, New York, NY, USA, 2009.

51. K. Sachs. *Performance Modeling and Benchmarking of Event-Based Systems*. PhD thesis, TU Darmstadt, 2011.
52. K. Sachs, S. Kounev, J. Bacon, and A. Buchmann. Benchmarking message-oriented middleware using the SPECjms2007 benchmark. *Performance Evaluation*, 66(8):410–434, Aug. 2009.
53. K. Sachs, S. Kounev, and A. Buchmann. Performance modeling and analysis of message-oriented event-driven systems. *Software and Systems Modeling*, pages 1–25.
54. A. Schröter, G. Mühl, S. Kounev, H. Parzyjegla, and J. Richling. Stochastic Performance Analysis and Capacity Planning of Publish/Subscribe Systems. In *4th ACM International Conference on Distributed Event-Based Systems (DEBS 2010), July 12-15, Cambridge, United Kingdom*. ACM, New York, USA, 2010. Acceptance Rate: 25%.
55. C. U. Smith. *Performance Engineering of Software Systems*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1990.
56. Sun Microsystems. Java Message Service (JMS) Specification - Ver. 1.1, 2002.
57. T. Verdickt, B. Dhoedt, F. Gielen, and P. Demeester. Automatic Inclusion of Middleware Performance Attributes into Architectural UML Software Models. *IEEE Transactions on Software Engineering*, 31(8):695–711, 2005.
58. D. Westermann, J. Happe, M. Hauck, and C. Heupel. The performance cockpit approach: A framework for systematic performance evaluations. In *Proceedings of the 36th EUROMICRO Conference on Software Engineering and Advanced Applications (SEAA 2010)*, pages 31–38. IEEE Computer Society, 2010.