Predictive Modelling of Peer-to-Peer Event-driven Communication in Component-based Systems

Christoph Rathfelder*[1], David Evans[2], and Samuel Kounev[3]

[1] Software Engineering
FZI Research Center for Information Technology
Karlsruhe, Germany
rathfelder@fzi.de
[2] Computer Laboratory
University of Cambridge
Cambridge, UK
david.evans@cl.cam.ac.uk
[3] Faculty of Informatics
Karlsruhe Institute of Technology (KIT)
Karlsruhe, Germany
kounev@kit.edu

Abstract. The event-driven communication paradigm is used increasingly often to build loosely-coupled distributed systems in many industry domains including telecommunications, transportation, and supply chain management. However, the loose coupling of components in such systems makes it hard for developers to estimate their behaviour and performance under load. Most general purpose performance meta-models for component-based systems provide limited support for modelling event-driven communication. In this paper, we present a case study of a real-life road traffic monitoring system that shows how event-driven communication can be modelled for performance prediction and capacity planning. Our approach is based on the Palladio Component Model (PCM) which we have extended to support event-driven communication. We evaluate the accuracy of our modelling approach in a number of different workload and configuration scenarios. The results demonstrate the practicality and effectiveness of the proposed approach.

1 Introduction

In event-driven component-based systems, system components communicate by sending and receiving events. Compared to synchronous communication using, for example, remote procedure calls (RPCs), event-driven communication among components promises several benefits [1]. For example, being asynchronous in nature, it allows a *send-and-forget* approach, i.e., a component that sends a

* This work was supported by the European Commission (Grant No. FP7-216556)

message can continue its execution without waiting for the receiver to acknowledge the message or react on it. Furthermore, the loose coupling of components provides increased flexibility and better scalability.

However, the event-driven programming model is more complex as application logic is distributed among multiple independent event handlers and the flow of control during execution can be hard to track. This increases the difficulty of modelling event-driven component-based architectures for performance prediction at system design and deployment time. The latter is essential in order to ensure that systems are designed and sized to provide adequate quality-of-service to applications at a reasonable cost. Performance modelling and prediction techniques for component-based systems, surveyed in [2], support the architect in evaluating different design alternatives. However, most general purpose performance meta-models for component-based systems provide limited support for modelling event-driven communication. Furthermore, existing performance prediction techniques specialised for event-based systems (e.g., [3]) are focused on modelling the routing of events in the system as opposed to modelling the interactions and message flows between the communicating components.

In [4, 5], we described an extension of the Palladio Component Model [6] that provides native support for modelling event-based communication in component-based systems. The Palladio Component Model (PCM) is a design-oriented performance meta-model for modelling component-based software architectures. It allows explicit capture of component context dependencies (e.g., dependencies on the component usage profile and execution environment) and provides support for a number of different performance analysis techniques. Based on our approach in [4], we developed a model-to-model transformation from the extended PCM to the original PCM allowing the use of existing analytical and simulative analysis techniques that significantly reduce modelling effort and complexity.

In the above publications, we briefly described the proposed PCM extension and model transformation with no validation of their effectiveness and accuracy. In this paper, we apply our modelling approach to a case study of a real-life road traffic monitoring system in order to validate its practicality, effectiveness and accuracy. The system we study is developed as part of the TIME project (Transport Information Monitoring Environment) [7] at the University of Cambridge. The system is based on a novel component-based middleware called SBUS (Stream BUS) [8] which supports peer-to-peer event-based communication including both continuous streams of data (e.g., from sensors), asynchronous events, and synchronous RPC. The contributions of this paper are: i) a refinement of our model transformation described in [4] to reflect the characteristics of the SBUS framework, ii) a case study of a real-life system showing how our approach can be applied to model event-driven communication, and iii) a detailed evaluation of the model accuracy in a number of different scenarios representing different system configurations and workloads.

The remainder of this paper is organised as follows. Sect. 2 introduces the PCM, which are the basis of the performance model and the SBUS framework, which is the communication middleware used within the case study. Sect. 3

presents the case study a traffic monitoring system and the resulting performance model followed by a detailed experimental evaluation of the model predictions in Sect. 4. Next, we present an overview of related work and finally in Sect. 6 we conclude with a brief summary and a discussion of ongoing and future work.

2 Foundations

In this section, we briefly introduce the Palladio Component Model that our modelling approach is based on. Furthermore we present an overview on the middleware SBUS (Stream BUS), which is the foundation for the traffic monitoring system presented as case study in Sect. 3.

2.1 Palladio Component Model

The Palladio Component Model (PCM) [6] is a domain-specific modelling language for modelling component-based software architectures. It supports automatic transformation of architecture-level performance models to predictive performance models including layered queueing networks [9], stochastic process algebras [10], and simulation models [6, 11]. In PCM, architectural models are parametrized over the system usage profile and the execution environment. This allows reuse of models in different contexts for different usage scenarios and execution environments.

Software components are the core entities of PCM. They contain an abstract behavioural specification for each provided component service called the Resource Demanding Service Effect Specification (RD-SEFF). RD-SEFFs describe by means of an annotated control flow graph how component services use system resources and call external services provided by other components. Similar to UML activities, RD-SEFFs consist of different types of actions:

- **InternalActions** model resource demands and abstract from computations performed inside a component. To express the performance-relevant resource interaction of the modelled computations, an InternalAction contains a set of **ParametricResourceDemands**.
- **AquireAction** and **ReleaseAction** are used to acquire and respectively release a semaphore which can be used, for example, to model a thread pool.
- **ExternalCallActions** represent component invocations of services provided by other components. For each external service call, component developers can specify performance-relevant information about the service input parameters. External service calls are always synchronous in PCM, i.e., the execution is blocked until the call returns.
- **Loops** model the repetitive execution of a set of actions. A probability mass function specifies the number of loop iterations which can depend on the service input parameters.
- **Branches** represent "exclusive or" splits of the control flow, where only one of the alternatives can be taken. In PCM, the choice can be either probabilistic or it can be determined by a guard. In the former case, each

alternative has an associated probability determining the likelihood of its execution. In the latter case, boolean expressions based on the service input parameters determine which alternative is executed.
– **Forks** split the control flow in several parts that are executed in parallel. Usually, forks are asynchronous, i.e., the original control flow continues to execute directly after the parts are forked.

2.2 SBUS

The SBUS framework was designed to support distributed transport applications. Data are collected from multiple sources, are processed in ways that may or may not be envisaged by the data owners, and are presented to users in useful ways. All communication in the SBUS world is by means of events. Details of the system are given by Ingram [8]; what follows is a summary to make the paper self-contained. SBUS is shown schematically in Figure 1. The basic entity is

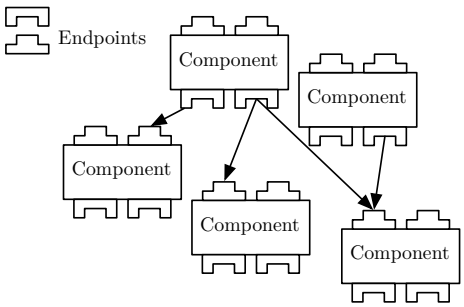


Fig. 1. The SBUS world.

the *component*. Components communicate via *messages* and this is how all data exchange is effected. Messages emanate from and are received by *endpoints*; each endpoint is plugged into one or more others. An endpoint specifies the schema of the messages that it will emit and accept. The framework enforces matching of sender and receiver schemas, ensuring that only compatible endpoints are connected. A system for polymorphic endpoints facilitates writing components that don't know ahead of time the schemas of the messages that they will produce or consume. The act of connecting two endpoints is called *mapping*. Each endpoint can be a *client*, a *server*, a *source*, or a *sink*. Clients and servers implement remote procedure call (RPC) functionality, providing synchronous request/reply, and are attached in many-to-one relationships. On the other hand, streams of events emitted from source endpoints are received by sinks. This communication is entirely asynchronous and attachment is many-to-many .

Each component is divided into a wrapper, provided by the SBUS framework, and the business logic that makes up the component's function. The wrapper manages all communication between components, including handling the network, registration of endpoints and management of their schemas, and reporting

on the component's status, including providing reflection. The separation of the wrapper is deliberate as it insulates business logic from dealing with unreliable network infrastructure as well as providing resilience in the face of failure of connected components. The business logic specifies its endpoints' mappings and the wrapper takes care of ensuring that these are carried out.

3 Case Study

The system we study is developed within the TIME project (Transport Information Monitoring Environment) [7] at the University of Cambridge. We first give an overview on the different components the system consists of. In the second part of this section, we present the performance model of this system. The prediction results and the validation is presented in Sect. 4.

3.1 TIME Traffic Monitoring System

The application estimates the speed of buses that are near traffic lights when they turn red. This application is interesting because it requires information describing the current state of traffic lights alongside location information from buses. These two sources of data are, in many cases, not maintained by the same organisation, meaning that our application must fuse data provided by multiple organisations. This is the type of environment for which SBUS was designed as it precludes a centralised approach. Our implementation of this application uses four classes of SBUS components (see Figure 2) described below. Due to the middleware SBUS, it is possible to distribute these components over several computing nodes as well as centralize them on one node without any changes of the components' implementation. Finding the maximal processable event rate for a given deployment option or a resource-efficient deployment scenario that still meets all requirements regarding the event processing times is a complex task. Using performance prediction techniques eases the analysis of performance attributes for different deployment scenario and event rates without prototypical implementations or test environments.

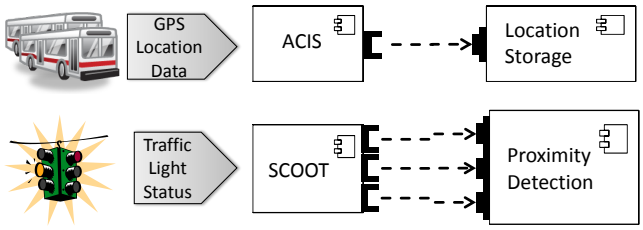


Fig. 2. The application's components.

Bus location provider (the "ACIS component") The bus location provider uses sensors (in our case, GPS coupled with a proprietary radio network) to note the locations of buses and report them as they change. Such a component produces

a stream of events, each containing a bus ID, a location, and the time of the measurement.

Location storage The location storage component maintains state that describes, for a set of objects, the most recent location that was reported for each of them. The component has no knowledge of what the objects are—each is identified only by name. The input is a stream of events consisting of name/location pairs with timestamps, making a Bus Location Provider a suitable event source. The location state is not conceptually a stream of events so, in our implementation, it is stored in a relational database that other components may query.

Traffic light status reporter (the "SCOOT component") The city of Cambridge, UK provided the testbed for our application. The city's traffic lights are controlled by a SCOOT system [12], designed to schedule green and red lights so as to optimise use of the road network. As a necessary part of controlling the lights, SCOOT knows whether each light is red or green[4] and can transmit a stream of information derived from vehicle detecting induction loops installed in the road. This component 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 reflect SCOOT's measurements of traffic flow, and two RPC endpoints that allow retrieval of information about a junction (such as its name and its location) and links between junctions (the junction the link attaches to, the location of the link's stop line, and so on).

Proximity detector This is the only application-specific component in our system. It receives a stream of trigger 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. The identities of these buses are then stored again in the relational database for use by the user interface; these are removed in response to a second stream of trigger events indicating when lights turn from red to green. No events are emitted by the proximity detector component because the user interface does not communicate using SBUS.

3.2 Performance Model

The current version of PCM only supports synchronous call-return communication between components. As we demonstrated in [4], it is possible to model asynchronous communication using a combination of non-synchronised fork actions and external service calls. In this paper, we use these extensions of PCM to model the TIME traffic monitoring application described in the previous section. We have refined the model transformation proposed in [4] to reflect the characteristics of the SBUS framework.

[4] Amber is not under the control of SCOOT but is managed by hardware near each light.

As shown in Figure 3, each event source is substituted by an SBUS-specific component, the `SBUSSourceEP`, which provides the interface `I_SBUSEventSourceEP` and requires the interface `I_SBUSEventSinkEP` once for each connected event sink. The event sinks of the receiving components are substituted by `SBUSSinkEP` components. The latter provide the `I_SBUSEventSinkEP` interface and require the `I_SBUSEventSourceEP` interface, respectively. The behaviour of the two

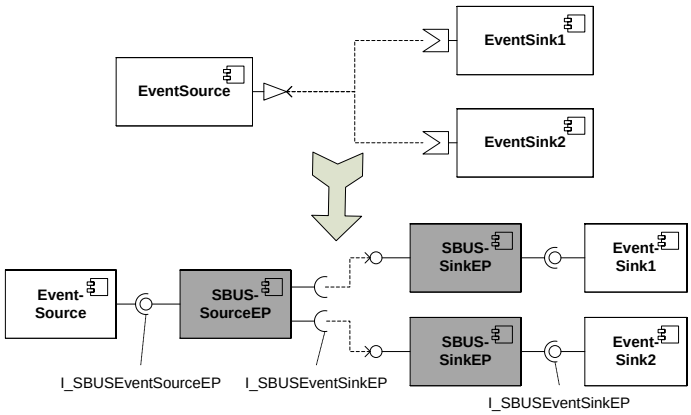
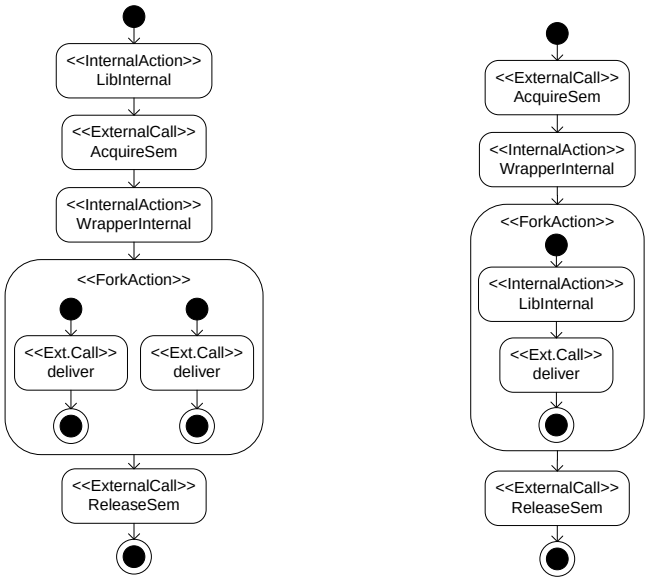


Fig. 3. Integration of SBUS-specific components.

SBUS-specific components is specified using RD-SEFFs. The RD-SEFF describing the `emit` method, which is part of `I_SBUSEventSourceEP`, is depicted in Figure 4(a). After an InternalAction modelling the CPU usage induced by the SBUS library, a semaphore encapsulated in a separate component is acquired. This semaphore reflects the single threaded behaviour of the SBUS wrapper. The wrapper's internal resource consumption is modelled with a second InternalAction followed by a ForkAction. The ForkAction contains a forked behaviour, which includes an ExternalCallAction for each connected `I_SBUSEventSinkEP` interface.

Similarly to `SBUSSourceEP`, the RD-SEFF of the `deliver` method, which is part of the `I_SBUSEventSinkEP` interface, includes actions to acquire and release a semaphore representing the single threaded implementation of the SBUS wrapper. The complete RD-SEFF of this method is illustrated in Figure 4(b). After an InternalAction representing the CPU usage by the SBUS wrapper, the `deliver` method of the connected component is called. As the component runs in a separate process from the wrapper, the call of the component and the InternalAction representing the CPU usage induced by the library are encapsulated in a ForkAction.

As discussed in Sect. 2.2, the SBUS framework also supports client-server communication following an RPC style. Therefore, we also add an SBUS-specific component for each client-server interface provided or required by a component in our scenario. The respective RD-SEFFs look very similar to those already described, however, they do not include the ForkAction used to model asynchronous control flow. Figure 5 illustrates the resulting PCM model showing the



(a) RD-SEFF of SBUSSourceEP (b) RD-SEFF of SBUSSinkEP

Fig. 4. SBUS-specific behaviour of components.

components of the TIME scenario in grey and the SBUS-specific components in white.

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 10000 measurements whose variation was negligible. The results, shown in Table 1, were used as estimates of the respective resource demands.

Component	Endpoint	Time in Component	Time in Library	Time in Wrapper
ACIS	feeds	0,5172 ms	0,0369 ms	0,0097 ms
Location	feeds	0,6343 ms	0,0088 ms	0,0088 ms
SCOOT	lightred	0,6266 ms	0,0400 ms	0,0167 ms
	lightgreen	0,6266 ms	0,0400 ms	0,0192 ms
	linkinfo	0,5225 ms	0,0180 ms	0,0137 ms
Proximity	lightred	0,4511 ms	0,0005 ms	0,0072 ms
	lightgreen	0,3139 ms	0,0005 ms	0,0072 ms
	linkinfo	0,0000 ms	0,0090 ms	0,0197 ms

Table 1. Results of resource demand estimation experiments.

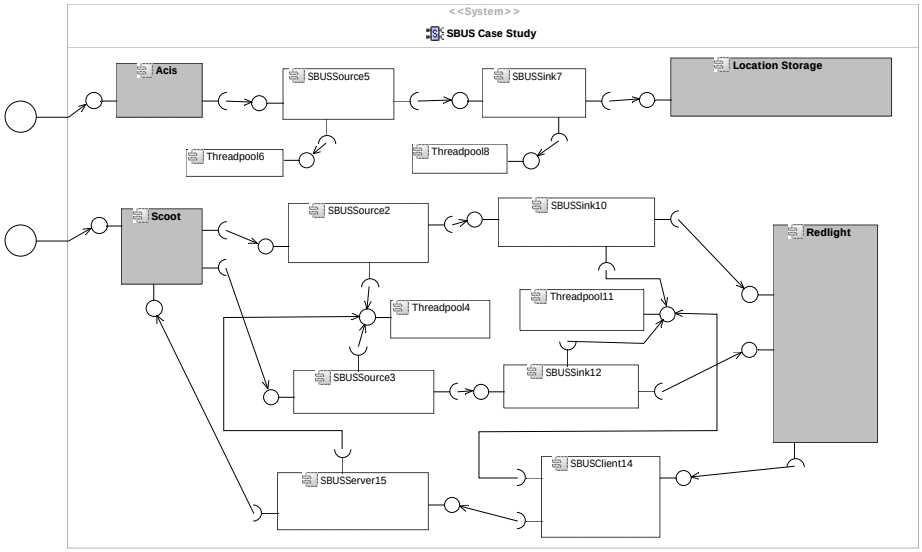


Fig. 5. System view of the PCM model.

4 Experimental Evaluation

Our experimental environment consisted of 12 identical machines, each equipped with a 2.4 GHz Intel Core2Quad 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 to a GBit LAN. The experiments were executed on one or more of these systems. In order to validate the developed performance model, we considered several different deployment scenarios of the TIME traffic monitoring application. For each scenario, we run multiple experiments under increasing event rates resulting in increasing utilisation levels of the system. We then compared the model predictions against measurements on the real system in order to evaluate the model accuracy. The selection of different deployment scenarios allows us to separate different possible influence factors like single-threaded implementations of components or influences from concurrently running component instances. These influence factors are then combined in the later scenarios. In the following, we summarise the results for each of the four scenarios we considered.

Scenario 1 As described in Sect. 3, the SCOOT component is connected to the Proximity Detector and the ACIS component is connected to the Location Storage component. To explore each of these interactions individually, we deployed ACIS together with Location Storage on one machine and SCOOT with Proximity Detector on another one. In Tables 3 and 2 we show the measurements of CPU utilisation[5] and compare them with the model predictions. The results are visualised in Figure 6. The prediction error is below 10% in most of the cases with exception of the cases under very low CPU utilisation. However, in these

[5] The CPU utilisation shown is over all four cores of the respective machine.

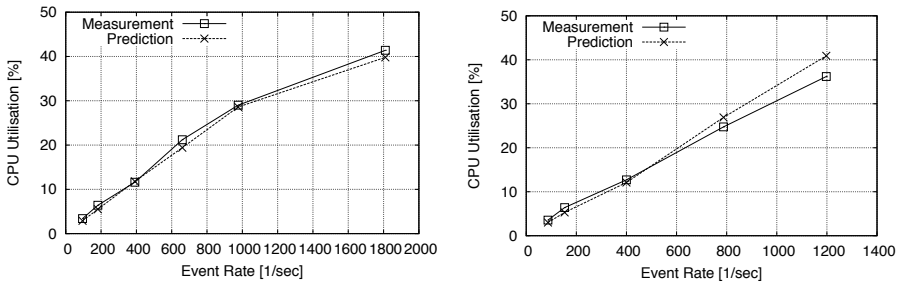
cases, the error is only 1% when considered as an absolute value which is negligible and can be explained by normal OS tasks. Note that each component has two threads, one executing the business logic and one executing the wrapper[6]. However, given that the resource demands of the wrapper are very low, most of the processing time is spent in the thread executing the business logic such that in practice only one thread per component is active most of the time. This explains why the CPU utilisation did not exceed 50% since each machine was running only 2 components while it had 4 cores available.

Event rate $[1/sec]$	86,82	153,36	399,88	787,32	1197,6
Measurement	3,55%	6,4%	12,7%	24,75%	36,2%
Prediction	3%	5,3%	12,1%	26,9%	40,9%
Error	15,49%	17,19%	4,72%	8,69%	12,98%

Table 2. Scenario 1: CPU Utilisation (SCOOT and Proximity Detector).

Event rate $[1/sec]$	95,05	182,02	391,45	660,18	976,4	1809,9	1880,2	1917,85
Measurement	3,4%	6,4%	11,6%	21,2%	29%	41,4%	45,2%	44,4%
Prediction	2,9%	5,5%	11,8%	19,4%	28,6%	39,8%	45,8%	47,9%
Error	14,71%	14,06%	1,72%	8,49%	1,38%	3,86%	1,33%	7,88%

Table 3. Scenario 1: CPU Utilisation (ACIS and Location Storage).



(a) ACIS and Location Storage (b) SCOOT and Proximity Detector

Fig. 6. Scenario 1: CPU Utilisation.

Scenario 2 The first scenario did not include CPU contention effects since there were more CPU cores than active threads. In this scenario, we use the same setup as before however with multiple instances of the components. Each component has three instances which results in 6 active threads per machine. In addition to considering CPU utilisation, this time we also analyse the effect of CPU contention on the event processing times. Tables 4 and 5 compare the predicted CPU utilisation against the measured CPU utilisation. The event rates

[6] In reality, each component has a separate thread for each endpoint. Thus, the SCOOT component actually has 4 threads: 1 for the wrapper and 3 for the three endpoints it provides. However, since the 2 RPC endpoints are not used that frequently, we only count the thread of the source endpoint.

listed there are per instance of the component and thus the overall processed event rate is three times higher. As we can see from the results, with exception of the cases under very low load, the modelling error was less than 5%.

We now consider the event processing times. We compare the measured processing time of the Location Storage component with the model predictions. The results are listed in Table 6 and visualised in Figure 7. As we can see, the model predictions are 5% to 10% lower than the measurements on the system.

Event rate $[1/sec]$	95,05	182,02	391,45	660,18	976,4	1809,9
Measurement	9,7%	18%	35,6%	57,6%	80,59%	91,59%
Prediction	8,6%	16,4%	35,6%	58,3%	80,3%	92,5%
Error	11,34%	8,89%	0,00%	1,22%	0,36%	0,99%

Table 4. Scenario 2: CPU Utilisation (ACIS and Location Storage).

Event rate $[1/sec]$	86,96	154	398,6	792
Measurement	9,94%	16,95%	40,21%	82,37%
Prediction	8,90%	15,80%	40,80%	81,60 %
Error	1,14%	1,38%	0,98%	4,36%

Table 5. Scenario 2: CPU Utilisation (SCOOT and Proximity Detector).

Event rate $[1/sec]$	95,05	182,02	391,45	660,18	976,4	1809,9
Measurement $[ms]$	0,657	0,695	0,676	0,703	0,780	0,846
Prediction $[ms]$	0,634	0,634	0,634	0,634	0,700	0,855
Error	3,57%	8,71%	6,14%	9,88%	10,22%	1,06%

Table 6. Scenario 2: Mean processing time of Location Storage.

Scenario 3 The previous scenarios evaluated the SCOOT and ACIS interactions on separate machines. In this scenario, all four components are deployed on the same machine. Similarly to Scenario 1, we deployed only one instance of each component. The results for CPU utilisation are shown in Table 7. As previously, with exception of the cases under very low load, the modelling error was below 5%.

Even though in this case, we have 4 active threads (one per component), it was not possible to scale beyond a CPU utilisation of 75%. This is because the computational load is not spread uniformly among the four threads and they are not running independently of one another (i.e., the Proximity Detector component is triggered by SCOOT and the Location Storage component is triggered by ACIS). As a result of this, not all four threads are always active at the same time and the 4 CPU cores cannot be saturated. The model predictions for the event processing times were of similar accuracy to the ones shown in the previous scenario, so we omit them here.

Scenario 4 In this last scenario, similarly to the previous one, we again deployed all components on one machine, however, this time we used two instances of each

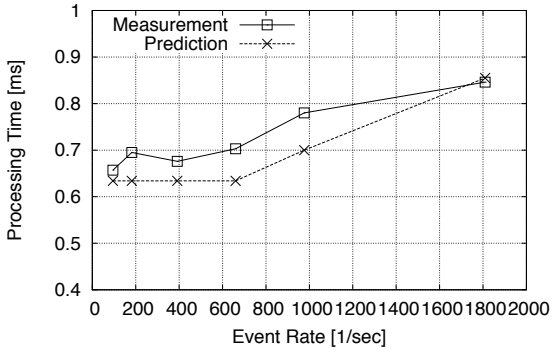


Fig. 7. Scenario2: Mean processing time of Location Storage.

Event rate ACIS [1/*sec*]	94,81	180,05	393,25	940,10	1328,39	1144,95	1084,29
Event rate SCOOT [1/*sec*]	106,80	153,18	396,39	776,68	777,15	943,15	1145,35
Measurement	6,81%	11,90%	26,50%	53,66%	63,78%	65,14%	70,81%
Prediction	6,50%	10,70%	25,30%	55,10%	66,40%	67,40%	74,20%
Error	4,50%	10,07%	4,53%	2,68%	4,11%	3,47%	4,79%

Table 7. Scenario 3: CPU Utilisation.

component. The latter results in 8 active threads processing business logic of the four components. The results for CPU utilisation are shown in Table 8. As we can see, the higher number of threads allows to saturate the machine. Compared to the previous scenarios, the prediction error is slightly higher, however, it is still mostly below 10%. Again, the model predictions for the event processing times were of similar accuracy to the ones shown in Scenario 2, so we omit them.

Event rate ACIS [1/*sec*]	94,57	178,30	578,50	711,74	764,65
Event rate SCOOT [1/*sec*]	86,76	153,10	394,38	545,19	609,98
Measurement	12,81%	23,04%	61,91%	82,94%	93,55%
Prediction	11,7%	21,4%	65,8%	91,4%	98,5%
Error	8,65%	7,12%	6,28%	10,20%	5,30%

Table 8. Scenario 4: CPU Utilisation.

In summary, the developed model proved to capture the system behaviour well and to provide accurate performance predictions under varying configurations and deployment scenarios. With a few exceptions, the modelling error was mostly below 10%. Using the model, we were able to predict the CPU utilisation for a given event rate as well as the maximum event rate that can be sustained in a given deployment. In many cases, the maximum CPU utilisation that could be reached was lower than would be expected due to the uneven distribution of the computational load among the active component threads. The model enabled us to accurately predict the maximum event rate that could be reached with a given number of component instances deployed on the physical machines. Furthermore, the model provided accurate predictions of the event processing times in scenarios with CPU contention. The developed model provides a tool for performance prediction and capacity planning that can be used to detect

system bottlenecks and ensure that the system is designed and sized to sustain its expected workload satisfying performance requirements.

5 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 including message-oriented middleware.

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) [13]. 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 [14] and its successor the UML MARTE profile [15], both of which are extensions of UML as the de facto standard modelling language for software architectures.

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 service-oriented systems. A recent survey of methods for component-based performance-engineering was published in [2].

Several approaches use model transformations to derive performance prediction models (e.g., [16–18, 6]). Cortellessa et al. surveyed three performance meta-models in [19] leading to a conceptual MDA framework of different model transformations for the prediction of different extra-function properties [20, 21]. The influence of certain architectural patterns in the system's performance and their integration into prediction models was studied by Petriu [17, 22] and Gomaa [23]. In [17, 22], UML collaborations are used to modelle the pipe-and-filter and client-server architectural patterns which are later transformed into Layered Queueing Networks.

In the following, we present an overview of existing performance modelling and analysis techniques specialized for event-based systems including systems based on message-oriented middleware (MOM). A recent survey of techniques for benchmarking and performance modelling of event-based systems was published in [24]. In [25], 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 [26]. 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-driven interactions and message flows.

A method for modelling MOM systems using performance completions is presented in [27]. Model-to-model transformations are used to integrate low-level details of the MOM system into high-level software architecture models. A case study based on part 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 [28], 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 modelling 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. However, again the workloads considered do not include multiple message exchanges or interaction mixes.

Several performance modelling techniques specifically targeted at distributed publish/subscribe systems exist in the literature. However, these techniques are normally focused on modelling the routing of events through distributed broker topologies from publishers to subscribers as opposed to modelling interactions and message flows between communicating components in event-driven applications. In [3] 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 [29, 24], a methodology for workload characterization and performance modelling 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.

6 Conclusions and Future Work

In this paper, we presented a case study of a real-life road traffic monitoring system showing how event-driven communication can be modelled for performance prediction and capacity planning by means of an extended version of the Palladio Component Model (PCM). We refined our PCM extension from [4] customising it to the specific middleware framework used in the considered system. We developed a performance model of the system and conducted a detailed

experimental evaluation of the model accuracy in a number of different scenarios representing different system configurations and workloads. The presented case study is the first validation of our modelling approach demonstrating its practicality, effectiveness and accuracy.

The results presented in this paper form the basis for several areas of future work. Currently, we are working on integrating the meta-model extensions into the Palladio tool chain and fully automating the proposed transformation. Furthermore, we plan to refine the transformation to separate general platform-independent event-based behaviour from platform-specific resource demands and behaviour. This separation will allow modelling of event-driven communication independently of the infrastructure used. The platform-specific resource demands will be added later using predefined extension points and completions. As a next step, we plan to work on extracting prediction models automatically at run-time. The resource discovery component (RDC) which is part of the SBUS framework provides methods to determine the connections between endpoints. This information can be used to create the system model. Additionally, we plan to extend the instrumentation we integrated in the SBUS framework making the measured resource demands available during operation. This will allow to extract model parameters dynamically at run-time and will make it possible to use the models for adaptive run-time performance management.

References

1. Hohpe, Gregor ; Woolf, B.: Enterprise integration patterns : designing, building, and deploying messaging solutions. 11. print. edn. The Addison-Wesley signature series. Addison-Wesley, Boston, Mass. (2008)
2. Koziolek, H.: Performance evaluation of component-based software systems: A survey. Performance Evaluation **67-8**(8) (2009) 634–658 Special Issue on Software and Performance.
3. Mühl, G., Schröter, A., Parzyjegla, H., Kounev, S., Richling, J.: Stochastic Analysis of Hierarchical Publish/Subscribe Systems. In: Proc. of Euro-Par 2009). (2009)
4. Rathfelder, C., Kounev, S.: Position Paper: Modeling Event-Driven Service-Oriented Systems using the Palladio Component Model. In: Proc. of QUASOSS 2009, ACM, New York, NY, USA (2009) 33–38
5. Rathfelder, C., Kounev, S.: Fast Abstract: Model-based Performance Prediction for Event-driven Systems. In: (DEBS2009), Nashville, TN, USA (July 2009)
6. Becker, S., Koziolek, H., Reussner, R.: The Palladio component model for model-driven performance prediction. Journal of Systems and Software **82** (2009) 3–22
7. Bacon, J., Beresford, A.R., Evans, D., Ingram, D., Trigoni, N., Guitton, A., Skordylis, A.: TIME: An open platform for capturing, processing and delivering transport-related data. In: Proceedings of the IEEE consumer communications and networking conference. (2008) 687–691
8. Ingram, D.: Reconfigurable middleware for high availability sensor systems. In: Proc. of DEBS 2009, ACM Press (2009)
9. Koziolek, H., Reussner, R.: A Model Transformation from the Palladio Component Model to Layered Queueing Networks. In: Performance Evaluation: Metrics, Models and Benchmarks, SIPEW 2008. Volume 5119. (2008)
10. Happe, J.: Predicting Software Performance in Symmetric Multi-core and Multiprocessor Environments. Dissertation, University of Oldenburg, Germany (2008)

11. Becker, S.: Coupled Model Transformations for QoS Enabled Component-Based Software Design. Volume 1 of Karlsruhe Series on Software Quality. Universitätsverlag Karlsruhe (2008)
12. Hunt, P.B., Robertson, D.I., Bretherton, R.D., Winton, R.I.: SCOOT—a traffic responsive method of coordinating signals. Technical Report LR1014, Transport and Road Research Laboratory (1981)
13. Smith, C.U.: Performance Engineering of Software Systems. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA (1990)
14. Object Management Group (OMG): UML Profile for Schedulability, Performance, and Time (SPT), v1.1 (January 2005)
15. Object Management Group (OMG): UML Profile for Modeling and Analysis of Real-Time and Embedded systems (MARTE) (May 2006)
16. Marzolla, M.: Simulation-Based Performance Modeling of UML Software Architectures. PhD Thesis TD-2004-1, Dipartimento di Informatica, Università Ca' Foscari di Venezia, Mestre, Italy (February 2004)
17. Petriu, D.C., Wang, X.: From UML description of high-level software architecture to LQN performance models. In Nagl, M., Schürr, A., Münch, M., eds.: Proc. of AGTIVE'99 Kerkrade. Volume 1779., Springer (2000)
18. Di Marco, A., Inveradi, P.: Compositional Generation of Software Architecture Performance QN Models. In: Proc. of WICSA 2004
19. Cortellessa, V.: 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, New York, NY, USA, ACM (2005) 195–204
20. Cortellessa, V., Pierini, P., Rossi, D.: Integrating software models and platform models for performance analysis. IEEE Trans. on Softw. Eng. **33** (2007) 385–401
21. Cortellessa, V., Di Marco, A., Inverardi, P.: Integrating Performance and Reliability Analysis in a Non-Functional MDA Framework. In: Proc. of Fundamental Approaches to Software Engineering (FASE) 2007. Volume 4422. (2007) 57–71
22. Gu, G.P., Petriu, D.C.: XSLT transformation from UML models to LQN performance models. In: Proc. of WOSP 2002, ACM Press (2002) 227–234
23. Gomaa, H., Menascé, D.A.: Design and performance modeling of component interconnection patterns for distributed software architectures. In: WOSP '00: Proceedings of the 2nd international workshop on Software and performance, New York, NY, USA, ACM (2000) 117–126
24. Kounev, S., Sachs., K.: Benchmarking and Performance Modeling of Event-Based Systems. it - Information Technology (5) (October 2009) Survey Paper.
25. Henjes, R., Menth, M., Zepfel, C.: Throughput Performance of Java Messaging Services Using WebsphereMQ. In: Proc. of ICDCSW '06. (2006)
26. Menth, M., Henjes, R.: Analysis of the Message Waiting Time for the FioranoMQ JMS Server. In: Proc. of ICDCS '06, Washington, DC, USA (2006)
27. Happe, J., Becker, S., Rathfelder, C., Friedrich, H., Reussner, R.H.: Parametric performance completions for model-driven performance prediction. Performance Evaluation **67**(8) (2010) 694 – 716 Special Issue on Software and Performance.
28. Liu, Y., Gorton, I.: Performance Prediction of J2EE Applications Using Messaging Protocols. Component-Based Software Engineering (2005) 1–16
29. Kounev, S., Sachs, K., Bacon, J., Buchmann, A.: 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)