Model-Driven Performance Evaluation for Service Engineering

Claus Pahl[1] and Marko Bošković[2] and Wilhelm Hasselbring[2]

[1] Dublin City University, School of Computing, Dublin 9, Ireland
cpahl@computing.dcu.ie
[2] University of Oldenburg, Software Engineering, D-26111 Oldenburg, Germany
[boskovic|hasselbring]@informatik.uni-oldenburg.de

Abstract. Service engineering and service-oriented architecture as an integration and platform technology is a recent approach to software systems integration. Software quality aspects such as performance are of central importance for the integration of heterogeneous, distributed service-based systems. Empirical performance evaluation is a process of measuring and calculating performance metrics of the implemented software. We present an approach for the empirical, model-based performance evaluation of services and service compositions in the context of model-driven service engineering. Temporal databases theory is utilised for the empirical performance evaluation of model-driven developed service systems.
Keywords: Service-oriented Architecture, Model-Driven Development, Performance Evaluation, Instrumentation.

1 Introduction

The complexity of software makes its development costly and error-prone. Model-driven engineering (MDE) is an approach to deal with complexity by making software models primary artefacts of the development process. A model is closer to the problem domain than to the underlying implementation. Therefore, it moves the focus of software engineering from technology-specific implementation to the problem domain. MDE utilises two aspects of models. Firstly, complete implementations can be generated from models, but more importantly here, predictions about a software system can be made based on a model. A fully model-based approach hide code-level details and allows a software architect to concentrate on design-stage artefacts.

To provide trustworthy software, quality attributes [4] have to be satisfied. Quality aspects have not been addressed in sufficient depth in the context of heterogeneous, distributed, service-based systems. Service engineering and service-oriented architecture as an integration and platform technology is a recent approach to service-based software system integration. Performance as one of quality attributes, defined as a degree to which a software system meet its objectives for timeliness [5], is of central importance in this context.

At present, research in model-driven performance engineering is mostly dedicated to simulation and performance prediction with mathematical analysis methods [6, 7]. Nevertheless, predictions have to be validated when a software system is implemented and deployed. Validation should be based on modelling constructs as predictions are made according to them. Currently, timing behaviour is analyzed based on source code constructs (e.g., method execution time). In MDE, the level of abstraction is raised. Consequently, observations should be based on modelling constructs, such as components and their states, activities, interactions or methods.

In software engineering, instrumentation is the process of adding software probes to the program [5]. Software probes are additional pieces of code for collecting data about the software execution. A model-based language for instrumentation needs to be derived. Instrumentation languages can enforce data collection in relational manner.

We investigate the empirical performance evaluation of model-driven service-based systems. We focus on composed (or orchestrated) services processes and address their performance behaviour. We present work-in-progress that comprises:

- an instrumentation notation for service models that allows specific service model elements such as services or composition and flow operators to be annotated and marked as providing performance-relevant time information at execution time. We use UML activity diagrams to express service compositions and base our instrumentation language on this UML diagram format. Our work follows others in using UML beyond classical software design. The Erickson-Penker Business Extensions for UML [2] for instance permits UML to document an entire business enterprise.
- model-driven transformation techniques that generate executable code including the monitoring instructions necessary to record time information.
- a trace analysis query language. This language provides the ability to calculate performance metrics such as response time and throughput. The evaluation is based on temporal databases theory [8]. The temporal databases theory relates facts stored in a relational manner with time information. A relational trace is a dynamic list of events and timing information generated by the program as it executes [9]. A query language allows the evaluation based on the traces in terms of service model elements.

Empirical code-level instrumentation and analysis has been investigated in depth. Simulation and analytical models have been used to provide support at design stages of the development process. Our contribution represents a novel approach for the empirical, model-level evaluation of performance for service-based software systems that

- firstly, an empirical and, thus, ultimately more accurate and reliable technique than simulation and analysis,
- secondly, a fully model-based evaluation technique for the architect that hides code-level details.

The paper is structured as follows. The next section gives an overview of model-driven engineering and service engineering. Motivation and foundations of performance engineering are presented in Section 3. Section 4 introduces the instrumentation language. Performance monitoring through code generation and instrumented execution is described in Section 5. The analysis of evaluation data is discussed in Section 6. Related work is discussed in Section 7 and Section 8 concludes the paper.

2 Model-Driven Development and Service Engineering

The general idea of model-driven engineering (MDE) is to introduce a model as a first-class entity. With models, the development focus is moved to the problem domain. Models often enable the exploitation of formal methods. With abstraction, the understanding of the problem and its realization can be improved. Often, a complete implementation can be generated [10]. Model Driven Architecture (MDA) [11] is one approach for MDE initiated by the Object Management Group (OMG), a consortium of software vendors and users. MDA is based on three ideas: direct representation to shift the focus of software development away from technology toward the problem domain, automation to mechanize the relation of semantic concepts of problem domain and implementation domain, and open standards to enable interoperability to close the semantic gap between domain problems and implementation technologies. Our aim is to enable the evaluation of service performance when the primary artefact is a service (or service process) model.

A service is defined as a piece of software, whose public interfaces are defined and described using an interoperable format. Other systems can interact with the service in a manner prescribed by its definition. The composition of services to orchestrated processes is a major concern in current Web service research [12, 13]. These recent developments have strengthened the importance of architectural questions such as service composition.

Modelling can support these architectural questions. Behaviour and interaction processes are central modelling concerns for service-based software architectures. Fig. 1 illustrates how a UML activity diagram can be used to express a service orchestration – at an abstract level without addressing individual service providers. Four services that provide an online bank account facility login, balance, transfer, and logout are orchestrated into a process starting with a login, then allowing a user to iteratively choose between balance enquiries and money transfers, before logging out.

Explicit models enable developers and clients of services to create reliable service architectures using tool support. A model-driven development approach can even support automated code generation and performance analysis. Assuming that concrete, provided services are already attached to each service element, then an executable WS-BPEL process for the Web service platform can be generated. As we are going to demonstrate, the service composition model can be

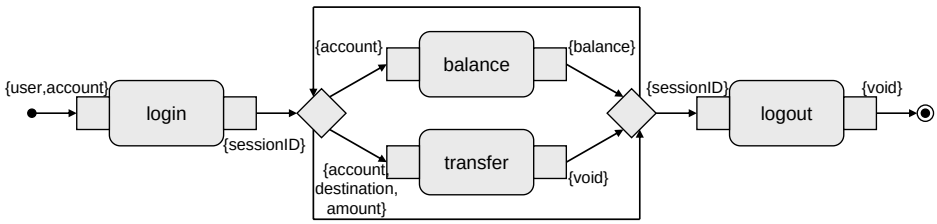

Fig. 1. Service Process modelled using a UML activity diagram.

instrumented for empirical performance analysis and executable processes including performance monitoring functionality can be generated.

3 Performance Evaluation

3.1 Software Performance, Evaluation and Motivation

Performance is considered as the degree to which a software system or component meets its objectives for timeliness [5]. It can be evaluated with simulation, analytical modelling or empirically [9]:

- Simulation is an imitation of a program execution focusing on specific aspects. It is less expensive than building a real system for empirical evaluation. It is flexible as changes can be dealt with easily if the simulation is derived automatically. However, simulation can suffer from a lack of accuracy.
- Analytical modelling is a technique where a system is mathematically described. Results of an analytical model can be less accurate than real-system measurements. However, analytical models are often easy to construct.
- Empirical evaluation is performed by measurements and metrics calculation. They provide the most accurate results as no abstractions are made.

The downside of performance evaluation by implementation, however, includes hardware dependency, extra cost of creating a prototype and deploying it, implementation deficiencies, and challenges in representative workload creation. Two observations led us to consider model-based empirical evaluations. Firstly, an approach for empirical evaluations of software performance for service-based software systems is still lacking despite its accuracy benefits. Secondly, empirical measurements and evaluations are currently performed only at the code level and mostly based on code constructs.

In model-driven engineering, observations of behaviour should be in terms of modelling constructs. Instrumentation for observing software should also be expressed in terms of these constructs in order to prevent the software architecture from having to represent transformation details and having to deal with code-level details. A necessary part of empirical performance evaluation is the execution data collection through instrumentation.

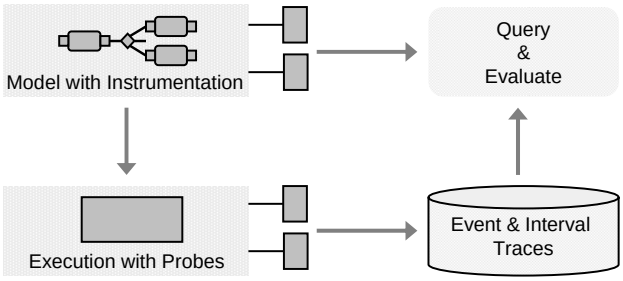

Fig. 2. Overview of the Framework.

3.2 Instrumentation

Instruments and instrumentation are commonly used for observing system behaviour and evaluating system properties in a range of disciplines. In software engineering, instrumentation is the process of adding software probes to a program [5]. Software probes are pieces of code for collecting data about the software execution. Two techniques for data collection exist:

- Sampling is a technique where parts of a program are sampled during its execution in some time interval - an example is sampling the program stack to follow program execution. It is a statistical technique in which a representative sample of data about the execution is taken. An advantage is that the impact on the performance of the program does not depend on the execution of the program. However, collected samples are different from run to run. The possibility that infrequent events are missed is another drawback.
- Event tracing is a process of generating traces of events in the software execution. A program trace is a dynamic list of events generated as the program executes [9]. A trace contains time-ordered events and can be used to characterize the overall program behaviour. Problems can be caused due to measurements. Each probe that is added causes execution overhead (performance) and event traces require resources (memory).

Due to its greater reliability, event tracing is used here. Event tracing is also more suitable for service-based software where the focus is on services as black-box entities that interaction in compositions. Traces are presented in our approach in relational manner using the concepts of temporal database theory to support the performance evaluation of traces.

3.3 Temporal Databases

Temporal databases support a notion of time [8]. In contrast to conventional databases, in which only facts are stored, each fact stored in a temporal database is associated with some time information. These facts can be related to a valid

time dimension and to a transaction time dimension. The valid time dimension is related to the time when the fact was true in reality. The transaction time dimension is related to the presence of the fact in the database.

Temporal databases which store only facts about the past are called historical databases [8]. Historical databases define two kinds of relations, event and interval relations [14]. Interval relations are used for storing facts which were true for some time interval. Event relations are used for storing facts which were true at some particular time point.

We utilise concepts from historical databases, such as both interval and event relations, to instrument service composition models.

4 Instrumentation

The execution of a program, such as execution and interactions of a composite service, can be characterized in terms of event and interval relations. For instance, if an element of a modelling language models a part of the program execution which lasts for some time interval, it can be instrumented by a specialization of the interval trace. Our instrumentation technique is developed around an instrumentation language, which is integrated with the service modelling language, i.e. is an extension of the UML activity diagrams that we use to model service orchestrations. Both service orchestration language and instrumentation language are presented at the meta-model layer. We present an overview of the approach in Fig. 2 that relates modelling and execution.

4.1 Service Process Meta-model

The banking example based on the orchestration of services to a service process from Fig. 1 is formulated in terms of a UML activity diagram. A (simplified) definition of UML activity diagrams as a process language is based on activity nodes and edges to represent services and their connectivity, respectively. We have given preference to UML activity diagrams over other process notations such as BPMN, because of UML's elaborate language extension mechanisms.

4.2 Instrumentation Meta-model

Our instrumentation notation comprises of two parts. Firstly, a basic trace package (Fig. 3) to capture the notion of traces, i.e. event and interval traces, and operations to capture these traces. Secondly, the instrumentation of activity diagrams using the MOF profile extension mechanism (Fig. 4).

The basic trace package reflects the required time dimensions and the recording concepts. The activity diagram instrumentation utilises these. This separation allows the basic instrumentation principles to be reused in a range of problem-specific or even model-specific circumstances – which is important as domain-specific languages are increasingly important. In the given instrumentation, actions as the central elements of activity diagrams and all control nodes

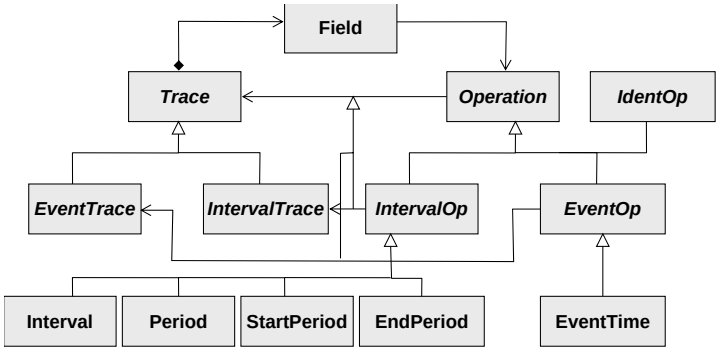

Fig. 3. Basic Trace Package.

are annotated. The execution of actions, which represent services at the model level, takes some time, i.e. an interval trace should be recorded at performance evaluation or execution time. We assume control flow decisions such as start and end of the overall process or choices and mergers as instantaneous events, i.e. modelled as event traces. This is a decision that can be modified at the Instrumentation Diagram level, without affecting the basic trace package. This provides for easy adaptability of the instrumentation to different interpretations and modelling languages.

4.3 Instrumentation Application

The application of the instrumented activity diagram is illustrated in Fig. 5. Two types of model elements - actions such as login or transfer and control nodes such as the start or the first decision point - are instrumented. An interval consisting of begin and end time of the service executions that implement the actions are recorded as a consequence of this instrumentation. Events, i.e. individual time stamps, are recorded for the control nodes.

For the service architect, it is import to find an adequate instrumentation that provides answers to the relevant performance questions. For instance, in a particular situation only the response times (average, maximum) of particular services, such as the account management services balance and transfer, are of interest. Then, the instrumentation needs to reflect these requirements.

While we consider this instrumentation of actions and control nodes to be the standard, the approach is flexible enough to accommodate context-specific customisations. Some control nodes could be excluded or other modelling elements could be added. This is only limited by the extent to which transformation and code generation support the different model element instrumentations. The instrumentation of elements could be disabled that are difficult to implement or whose analysis would not provide useful performance information.

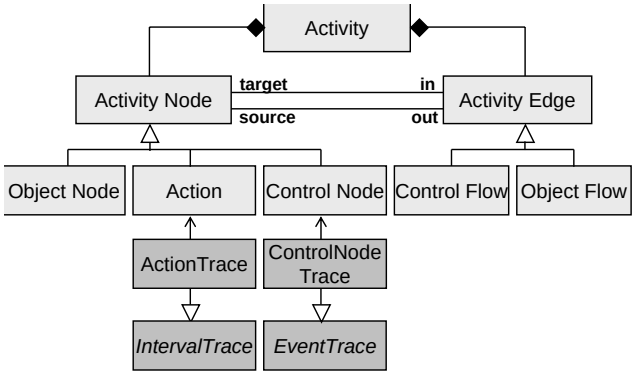

Fig. 4. Activity Diagram Instrumentation.

5 Performance Monitoring

The implementation of the instrumentation should be, firstly, easy to realise and, secondly, implemented without significant overhead. Aspects and interception techniques can be utilised to implement the instrumentation and data collection. Although an important aspect of performance evaluation, the focus of this paper is on the model-related issues of instrumentation specification (such as instrumentation meta-models) and data query and assessment activities (which are discussed later on). We only discuss principles of monitoring here.

5.1 Intrumentation and Monitoring

In the services context, often the problem arises that the addition of probes into the service implementation is not possible due to the nature of services as true black box software components. We therefore distinguish two scenarios:

- controlled environments that allow access to code. Aspect Oriented Programming (AOP) [15] is a programming approach, suitable for the controlled approach, which can be used for transparent software instrumentation. AOP is a technique that enables the separation of instrumentation from the development of the core software functionality [16, 17]. Marenholz et al. [17] use AspectC++ for the instrumentation of operating systems for debugging, profiling/measurement, and runtime surveillance/monitoring.
- open environments in which services are black-box components. For a transparent instrumentation of component systems, interceptors can be used. Interceptors are similar to AOP and can intercept method invocations to transparently instrument a program [18, 19]. Software probes can be predefined and placed in stubs and skeletons during an interface description compilation [19]. The probes can be turned on and off at runtime.

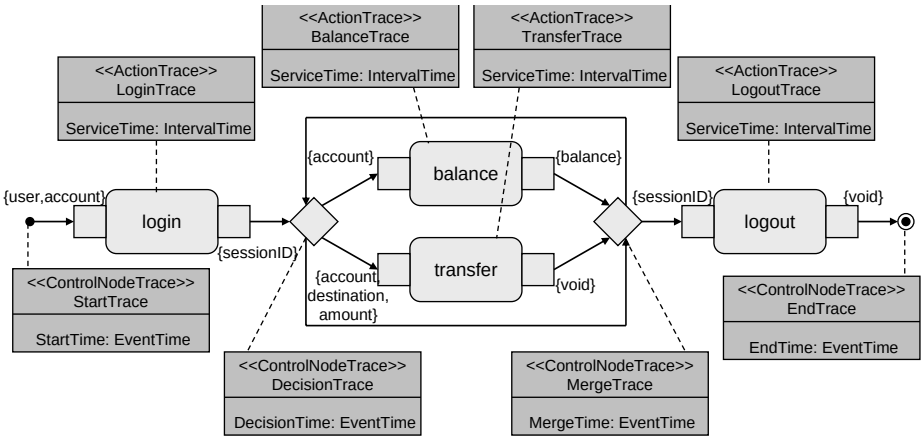

Fig. 5. Application of the Instrumentation to the online banking service process.

The JBoss Application Server, for instance, enables the transparent aspect-oriented addition of functionality. Its AOP features allow the interception of events and addition of trigger functionality based on those events.

5.2 Generation of Instrumented Code

Aspect Oriented Programming, interceptors, and bytecode and platform instrumentation are approaches that enable the collection of data without affecting the functionality. We utilize these ideas to collect data about the software execution at the model level as a separate concern.

The first step, however, is the generation of executable and instrumented code. Activity diagrams that model service orchestrations can be converted into executable Web services processes, if invocation information such as the service location is added to the abstract service process description:

- AOP concepts are used to generate the instrumented executable service code.
- Interception mechanisms add instrumentation and data collection.

We propose ATL transformations – the ATLAS Transformation Language ATL is a tool-supported hyrbid model transformation language for the ecplise platform – to transform activity diagrams into AOP-based code.

5.3 Performance Data Recording

The instrumentation includes monitoring and data collection functionality. Data is stored in a historical database such as TimeDB or temporal features in Oracle database servers. Fig. 6 shows a sample recording for the composite process for online banking based on the instrumentation defined in Fig. 5. Here, only data for the decision node and the transfer service are provided as samples.

TransferTrace: ServiceTime:	IntervalTime
2:22	2:45
3:03	3:12
3:15	3:29

DecisionTrace: DecisionTime: EventTime
2:19
2:50
3:01
3:10
3:35

Fig. 6. Collected Data for Online Banking Process Instrumentation.

6 Performance Evaluation

6.1 Analysis Language

Temporal and historical databases – which provide the conceptual background for the analysis part of our evaluation technique – are usually extensions of traditional relational databases. SQL is therefore available as a query language to retrieve information in relation to the recorded event and interval times and to use the language for common statistical operations.

The objective is to extract performance-relevant information from the basic times stored in a historical database that allows a software architect to assess the overall performance of individual services and also orchestrated processes. SQL is sufficient as a query language to formulate the relevant performance assessment queries. More advanced solutions like data warehouses with their extended evaluation support are not required. We can classify performance assessments as follows:

- Response time assessment: response times of activities are usually recorded as intervals. The SQL aggregate functions, such as average AVG or maximum MAX, provide the relevant answers.
- Frequency and distribution of invocations: the distribution of invocations (workload) between the individual services can be determined based on the calculation of ratios between total numbers of invocations.

The database representation directly reflects the modelling layer, as the representation is generated from the model instrumentation. The central goal of fully model-based performance evaluation is therefore achieved. The queries can consequently be formulated in terms of relevant model elements - which is one of the central objectives of model-driven quality engineering.

6.2 Performance Analysis

We have already classified the different types of performance assessments in the previous section. We now illustrate these types.

The average response time for service 'transfer' can be determined as follows:

```
SELECT AVG(ServiceTime)
FROM   TransferTrace
```

The determination of the maximum time can be formulated analogously. In the SOA context, where individual services are often provided by external organisations, this information is usually part of contracts and service-level agreements.

The proportion of 'transfer' invocations based in relation to all user selections (decisions) can also be formulated:

```
SELECT COUNT(ServiceTime) / COUNT(DecisionTime)
FROM   TransferTrace, DecisionTrace
```

This would allow a software architect to judge the frequency of individual service activations in typical application scenarios.

7 Related Work

There are several approaches for analytical evaluations of software performance from annotated models [7, 20] and simulation [21, 26]. A detailed survey related to performance prediction can be found in [6]. There are also several approaches for measurement and instrumentation in the context of code-centric development. Our contribution is an empirical instead of analytic technique for model-level evaluation.

- Snodgrass [22] introduces a relational approach for monitoring systems. His work shows that a relational data structure can be an appropriate formalism for monitoring dynamic behaviour of a system. The programmer manually defines the instrumentation according to concepts of an existing system. Our approach provides a schema for the definition of instrumentation languages according to the modelling formalism used for the specification of programs.
- Liao et al. [23] introduce a high-level language for program instrumentation and monitoring. A programmer specifies monitoring and measuring information, based on which a static analysis of code is done and instrumentation is added. However, their language is suited only for procedural languages.
- Another language for program instrumentation is the Metric Description Language (MDL) [24]. MDL has the ability to define instrumentation as a separate concern, independent of the program functionality, define points at which measurement actions should take place and weave these into a program at runtime. However, it is limited to functionally decomposed systems.

The idea on integrating software models and instrumentation is introduced in [25]. The authors develop a set of tools for a model-driven instrumentation. They define different program models such as a functional program model, a functional implementation model, a performance model and a monitoring model. Based on the monitoring model, the source code is instrumented and trace descriptions are generated. In our approach, the primary artefact of software development is

a model. Therefore, instrumentation is done at the model level. The functional implementation model is actually a product of software development. Furthermore, instrumentation defines what to measure and where to measure, and from these two models, automatically source code is produced.

In the specific context of service-based modelling, a lack of performance analysis is even more evident, although the need to address performance is recognised [26] and some solutions exist. The Application Response Measurement (ARM) standard [1] enables the collection of performance data. It is a conceptual library suited for usage with programming constructs. We relate our data collection with modeling constructs on a higher level of instrumentation. The standard only enables the collection of data, but not a systematic way of analyzing. ARM is only an interface for measurement. We introduce a systematic approach for data analysis as well as for instrumentation. A different architectural approach is taken in workflow management contexts [3]. Techniques in available workflow management languages and systems exist that provide timers, which allow to measure the invocation times out-of-the box by the workflow engine itself. Our solution adds a summarization component that performs arithmetic functions over a configurable period of time in a different architectural setting.

8 Conclusions

Empirical performance evaluation enables the validation of timeliness of a software system. In particular in service-oriented architecture, where software quality is paramount, the empirical approach that evaluates concrete application platforms is promising. However, currently an approach for empirical performance evaluation in the service development process where a model is the primary software artefact, is lacking. In order to provide software architects with tools for the reliable evaluation of performance, which goes beyond the predictive approaches of simulation and abstract analysis, a fully model-based instrumentation and analysis technique is necessary. The benefit of an empirical technique over predictive approaches is increased accuracy and therefore reliability of the evaluation results.

We have presented a service-specific approach for the empirical performance evaluation of model-driven developed services. Instrumentation and empirical performance evaluation is at the moment done based on programming language constructs at the source code level. Instrumentation at source code level for data collection about program executions in terms of modelling elements can be error-prone and can require significant effort. Therefore, the instrumentation needs to be done at the model level. The models for software functionality definition and instrumentation definition are separated to reduce the complexity of models.

Our contribution is based on a basic package for the definition of instrumentation languages for UML-based activity diagrams to model service orchestrations, a methodology for deriving instrumentation languages, and a query language for performance metrics calculation. The instrumentation languages enable automatically generated data collection in terms of modelling language constructs,

and are stored in the format of relational traces. Temporal database theory provides the background for the monitoring and analysis elements of the evaluation technique.

The instrumentation language is designed to be generic. The basic instrumentation package is application-independent and is, since it is separated from the application-specific instrumentation of specific model elements, transferable to other modelling notations and modelling domains. In order to demonstrate the flexibility of this approach, applications of our framework to class and state models are being investigated.

Currently, our generation and execution platform is not fully implemented. We plan to critically evaluate the feasibility of the approach by integrating software performance evaluation based on relational traces in some commercial tools for model-driven development. Furthermore, experiments on an extensive case study will be performed in order to show what the impact of instrumentation code and execution of the instrumented application is.

Another aspect specific to services shall be investigated in more detail. Our current work neglects specific issues arising from heterogeneous and fully distributed systems. The models we have considered here are activity diagrams that focus on the functional composition of services. The models used do not include the concept of distribution. Activity diagrams, however, allow modelling of distribution through activity partitions (so-called swimlanes). Since especially performance aspects apply to distributed service invocations, corresponding edges that cross partitions could be instrumented and the system could be monitored with respect to these inter-location invocations. We expect spatial-temporal databases to provide the foundations for this aspect.

References

1. Open Group. *Application Response Measurement - ARM*. 2002.
2. M. Penker and H.E. Eriksson. *Business Modeling With UML: Business Patterns at Work*. Wiley, 2000.
3. A. Kumar, W.M.P. Van Der Aalst, and E.M.W. Verbeek. Dynamic Work Distribution in Workflow Management Systems: How to Balance Quality and Performance, *Journal of Management Information Systems*, 18(3):157–193, 2002.
4. W. Hasselbring and R. Reussner. Toward trustworthy software systems. *Computer*, 39(4):91–92, 2006.
5. C.U. Smith and L.G. Williams. *Performance Solutions: A Practical Guide to Creating Responsive, Scalable Software*. Addison-Wesley, 2001.
6. S. Balsamo, A. Di Marco, P. Inverardi, and M. Simeoni. Model-Based Performance Prediction in Software Development: A Survey. *IEEE Transactions on Software Engineering*, 30(5):295–310, 2004.
7. Object Management Group. UML Profile for Schedulability, Performance, and Time Specification, OMG document formal/05-01-02. web: http://www.omg.org/cgi-bin/apps/doc?formal/05-01-02.pdf, 2005.
8. C. Zaniolo, S. Ceri, C. Faloutsos, R. Snodgrass, V. S. Subrahmanian, and R. Zicari. *Advanced Database Systems*. Morgan Kaufmann Pubishers, 1997.

9. D.J. Lilja. *Measuring Computer Performance: A Practitioner's Guide*. Cambridge University Press, 2000.
10. B. Selic. The Pragmatics of Model-Driven Development. *IEEE Software*, 20(5):19–25, 2003.
11. Object Management Group. MDA Guide. web: http://www.omg.org/cgi-bin/doc?ormsc/06-06-02.pdf, 2006.
12. G. Alonso, F. Casati, H. Kuno, and V. Machiraju. *Web Services – Concepts, Architectures and Applications*. Springer-Verlag, 2004.
13. L. Bass, P. Clements, and R. Kazman. *Software Architecture in Practice (2nd Edition)*. SEI Series in Software Engineering. Addison-Wesley, 2003.
14. N.L. Sarda. Extensions to SQL for Historical Databases. *IEEE Transactions on Knowledge and Data Engineering*, 02(2):220–230, 1990.
15. G. Kiczales, J. Lamping, A. Menhdhekar, C. Maeda, C. Lopes, J.-M. Loingtier, and J. Irwin. Aspect-Oriented Programming. In *Proc. of European Conf. on Object-Oriented Programming*, volume 1241, pages 220–242. Springer-Verlag, 1997.
16. M. Debusmann and K. Geihs. Efficient and Transparent Instrumentation of Application Components using an Aspect-oriented Approach. In *14th IFIP/IEEE Workshop on Distributed Systems: Operations and Management (DSOM 2003)*, volume 2867 of *LNCS*, pages 209–220. Springer, 2003.
17. D. Mahrenholz, O. Spinczyk, and W. Schroeder-Preikschat. Program Instrumentation for Debugging and Monitoring with AspectC++. In *Proc. 5th Int. Symp. on Object-Oriented Real-Time Distributed Computing ISORC '02*, pages 249–256. IEEE Computer Society, 2002.
18. M. Debusmann, M. Schmid, and R. Kroeger. Measuring End-to-End Performance of CORBA Applications using a Generic Instrumentation Approach. In *ISCC '02: Proc. 7th Int. Symp. on Computers and Communications*, pages 181–186. IEEE Computer Society, 2002.
19. J. Li. Monitoring of Component-Based Systems. Technical Report HPL-2002-25, Imaging Systems Laboratory, HP Laboratories Palo Alto, 2004.
20. D.B. Petriu and M. Woodside. A metamodel for generating performance models from UML designs. In *Proc. Int. Conf. The Unified Modelling Language: Modelling Languages and Applications*, LCNS 3273, pages 41–53. Springer-Verlag, 2004.
21. D. Park and S. Kang. Design phase analysis of software performance using aspect-oriented programming. In O. Aldawud, G. Booch, J. Gray, J. Kienzle, D. Stein, M. Kandé, F. Akkawi, and T. Elrad, editors, *The 5th Aspect-Oriented Modeling Workshop In Conjunction with UML 2004*, 2004.
22. R. Snodgrass. A Relational Approach to Monitoring Complex Systems. *ACM Transactions on Computer Systems*, 6(2):157–196, 1988.
23. Y. Liao and D. Cohen. A Specificational Approach to High Level Program Monitoring and Measuring. *IEEE Trans. on Software Eng.*, 18(11):969–978, 1992.
24. J. K. Hollingsworth, O. Niam, B. P. Miller, Z. Xu, M.J.R. Goncalves, and L. Zheng. MDL: A Language And a Compiler For Dynamic Program Instrumentation. In *PACT '97: Proc. 1997 Int. Conf. on Parallel Archit. and Compil. Techniq.*, pages 201–213. IEEE Comp. Society, 1997.
25. R. Klar, A. Quick, and F. Soetz. Tools for a Model–driven Instrumentation for Monitoring. In *Proc. 5th Int. Conf. on Modelling Techniques and Tools for Computer Performance Evaluation*, pages 165–180. Elsevier, 1991.
26. M.B. Blake. A Lightweight Software Design Process for Web Services Workflows. *Proceedings International Conference on Web Services ICWS 2006*, pages 411–418, IEEE Computer Society, 2006.