Model Driven Performance Measurement and Assessment with MoDePeMART*

Marko Bošković[1] and Wilhelm Hasselbring[2]

[1] Athabasca University, Canada,
marko.boskovic@athabascau.ca,
[2] Software Engineering Group, University of Kiel, Germany,
wha@informatik.uni-kiel.de

Abstract. Software performance is one of important software Quality of Service attributes. For this reason, several approaches integrate performance prediction in Model Driven Engineering(MDE). However, MDE still lacks a systematic approach for performance measurement and metrics assessment. This paper presents MoDePeMART, an approach for Model Driven Performance Measurement and Assessment with Relational Traces. The approach suggests declarative specification of performance metrics in a domain specific language and usage of relational databases for storage and metric computation. The approach is evaluated with the implementation of a UML Profile for UML Class and State diagrams and transformations from profile to a commercial relational database management system.

1 Introduction

Increasing dependency on software systems, and consequences of their failures, raises the question of software system trustworthiness [1]. In order to use software systems as dependable systems, means for quantification, verification, and contractual trust of those systems are being invented.

Means of quantification, verification, and contractual trust have to be done for both, functional and non-functional requirements. Functional requirements define functionality which is the objective of the system. Non-functional requirements are constraints on system's functionality offered by the system like security, privacy, reliability, timeliness etc [2]. They are characteristics of functionality design and implementation, and often they are called quality requirements [1].

Some of the non-functional properties of a service, of particular interest to users, are often specified with the Quality of Service (QoS). Performance, is one of the QoS attributes. In this paper, performance is defined as degree to which objectives for timeliness are met [3]. It describes timing behavior of a software system and it is measured with metrics like throughput and response time.

* This work is supported by the German Research Foundation(DFG), grant GRK 1076/1

Significance of meeting non-functional requirements in trustworthy software systems development, requires addressing them in the early design phases, in parallel to functional requirements. For this reason, research in meting performance requirements in Model Driven Engineering (MDE) was mostly dedicated to performance predictions with analytical modeling and simulation, e.g. [4]. Performance measurement and empirical assessment of predicted values are left to be done with profiling tools, or various techniques of manual insertions of code for data collection and metrics computation. There is still not a model driven approach for performance measurement and assessment.

This paper shows an approach for **Mo**del **D**riv**e**n **P**erformance **M**easurement and **A**ssessment with **R**elational **T**races) called ***MoDePeMART***. The essence of the approach is: (1) declarative specification of measurement points and metrics in a domain specific language, (2) automatic generation of code for data collection, storage, and metrics computation, and (3) usage of Relational Database Management Systems (RDBMS) for performance data storage and computation.

The paper is structured as follows. Section 2 explains the need for a model driven approach for performance measurement and assessment. Measurement and assessment with **MoDePeMART** is depicted in Section 3. The metamodel which enables declarative specification of measurements and metrics computation is described in Section 4. The evaluation of the approach through the implementation as a UML Profile for UML Class and State diagrams and transformations to MySQL RDBMS is shown in Section 5. Section 6 contains the comparative analysis of the approach with other approaches for performance measurement and assessment. The limitations (assumptions) of the approach are specified in Section 7. Section 8 gives an outlook of the approach and the directions for the future work.

2 Motivation

MDE is a software engineering paradigm which suggests using models as the primary artifacts of software development. It relies on two basic principles [5]: abstraction and automation.

Abstraction suggests usage of Domain Specific Modeling Languages (DSMLs). DSMLs are specialized modeling languages for solving classes of domain problems. Users of DSMLs are experts of that domain. Accordingly, DSMLs contain concepts used by domain experts. With DSMLs domain experts specify solutions to domain problems without being distracted by implementation details.

Automation handles implementation. It suggests transformations of DSML models to implementations. This principle can be seen as one more level of compilation.

In such a development process performance analyst faces several problems when trying to measure and assess performance. First, the modeling language used for software functionality development might not support constructs needed for the performance measurement and assessment, such as routines for obtaining time. Second, even if it does, a performance analyst is not an expert in that mod-

eling language, and it might be difficult for him to use it. Finally, data collection and assessment at the platform level can be error-pronouns. In order to do it a performance analyst would have to know how the domain specific constructs are transformed to the platform. To remove these problems we suggest declarative specification of metrics of interest in DSML and automatic instrumentation and code generation, facilitated with the **MoDePeMART**approach and depicted in the next section.

3 MoDePeMART: Model Driven Performance Measurement and Assessment with Relational Traces

MoDePeMARTintegrates performance measurement and assessment in MDE, in such a way that it is transparent to the developer. The example on UML is described in Figure 1.

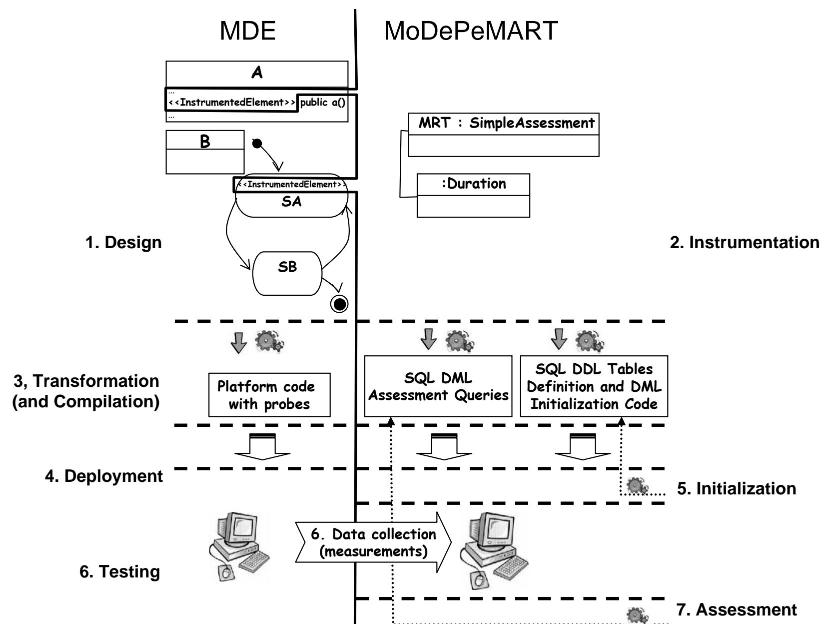

Fig. 1. Performance Measurement and Assessment in MDE with MoDePeMART

After the design model is finalized (1), the instrumentation (2) takes place. Here, measurement points are specified in the model. Furthermore, also are specified metrics of interest. Finally, the context of the service is specified. More on context is explained in Subsection 3.1. Measurement points, metrics, and context are specified in the DSML defined in Section 4.

From the design and performance measurement and assessment model transformation (3) generates software code with integrated code for performance data collection, and code for performance data storage and metrics computation. The generated code for performance data storage and metrics computation is SQL DDL code for tables needed for data storage, and SQL DML code for initial table entries required for metrics computation. The transformation is followed by compilation of the platform code.

After the deployment (5) of the generated code, RDBMS for storing and metrics computation is initialized (6). Next, execution of test cases takes place during which data about software execution are collected(7). Finally, to compute performance metrics SQL DML queries are executed (8).

MoDePeMARTapproach is language independent approach. However, it assumes some characteristics of modeling languages and systems. These characteristics and performance assessment in such systems are discussed in the next section.

3.1 Transformational and Reactive Software Systems and Performance Assessment

MoDePeMARTassumes that a modeling language for software development facilitates modeling of two subsystems: transformational and reactive. Transformational [6] systems are systems which take some input value and transform them to some output value through the set of steps specified by some algorithm. For the same input value, they will always go through the same steps. An example of transformational software system is in Figure 2 a).

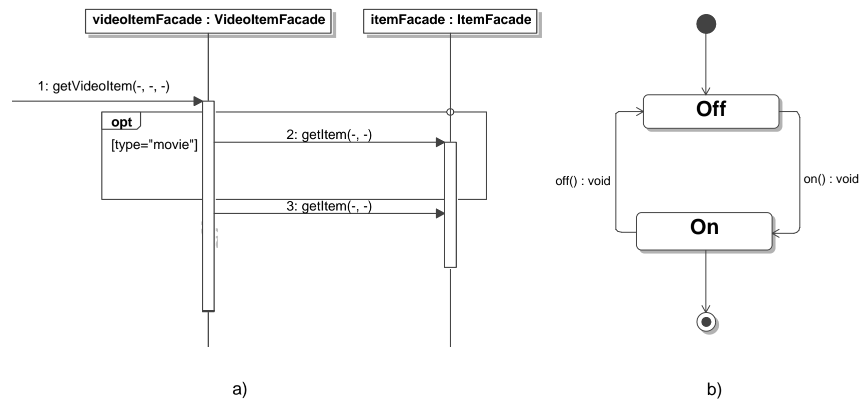

Fig. 2. Transformational (a) and reactive (b) behavior in **getVideoItem** method invocation.

The **getVideoItem** method is a method for obtaining video items in a small electronic items management application. Two kinds of items are obtained from the database with this method: a movie and a music video item. When user requests a movie, the value of the variable **type** is "movie" and the user gets two files: a movie trailer and the movie. The **getItem** invocation 2 obtains the trailer and the invocation 3 obtains the movie. When user requests a music video item, the value of **type** is not "movie" and only the **getItem** invocation 3 executes. This invocation obtains a music video file from the database.

Transformational programs are composed of [7]: simple commands (e.g. assigning a value to a variable), composite commands (e.g. a command block), guarded command (e.g. a UML option block or *if* statement), guarded command set (e.g. UML alternatives or the **C** *switch* statement), and loops. These commands are composed with two relations [7]: invocation (one uses another one) and sequential composition (one executes before another one).

Reactive software systems are systems which receive stimuli from environment and either change internal state, or produce some action in environment. The behavior depends on both stimulus and current system state. The reactive subsystem of the **ItemFacade** manages the data compression in database and the **getItem** method communication. When the state is **On**, the data is compressed in the DBMS and decompressed at the **ItemFacade** side. When **Off**, there is no compression.

The context of the service execution has to be taken into account when assessing performance. Inappropriate context specification can lead to inappropriate performance assessment. In systems with interwoven transformational and reactive part, both, transformational and reactive context have to be taken into account. Transformational context is the sequence of method (non)executions before and after the required service. For example, let us assume that it is of interest the response time of the **getItem** method when obtaining a movie file. If only the execution of the **getItem** would be considered without any specification of previous executions, the computed response time would also include executions of the **getItem** outside of the **getVideoItem** method. One more attempt without the specification of context is to consider the time between the invocation of the 3. **getItem** method from the **getVideoItem** method and the arrival of it's return value. However, in this case the final response time includes obtaining movies and music videos. The solution is in specification that the response time is computed for **getItem** method which is invoked from the **getVideoItem** method and that the optional block did not execute before the **getItem** execution.

Reactive context is the state of the system. The state can have a diverse impact on response time. For example, if the communication in the previous example is compressed, obtaining a movie response time can be reduced. However, the response time of obtaining a trailer can be increased. Due to the small size of the trailer the compression, transfer of compressed data, and decompression can take more time than transfer of non-compressed data.

4 The Metamodel for Performance Measurement and Assessment

In previous section it is explained that the **MoDePeMART**suggest declarative specification of performance measurements and metrics computation with a DSML. For this reason, the DSML defining metamodel facilitates the declarative specification of: execution context, and metric computation.

The declarative specification of transformational execution context is enabled with the part of the metamodel in Figure 3.

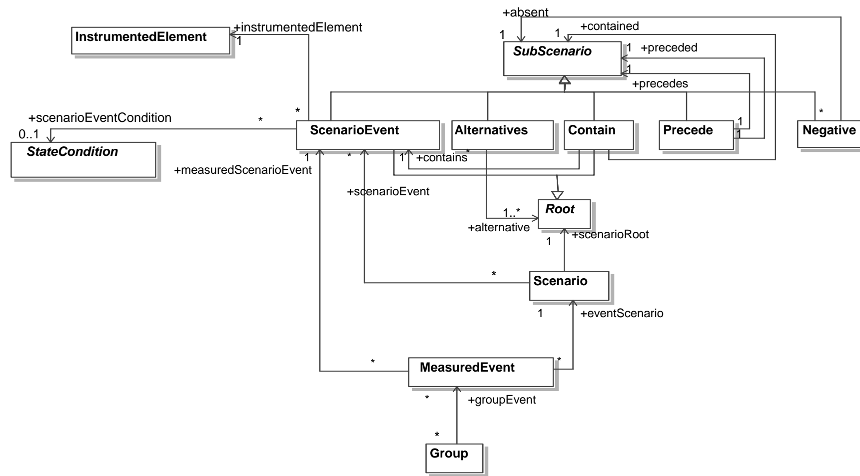

Fig. 3. The part of the metamodel for transformational context specification.

The measurement points of a model are specified with instances of the **InstrumentedElement** metaclass. Instrumented elements can be either simple commands or statement block.

Transformational context can be specified with instances of **Scenario**, **Root**, **ScenarioEvent**, **Alternatives**, **Contain**, **Precede**, **Negation** and **SubScenario** metaclasses. A transformational context is encapsulated in the **Scenario** metaclass, and consists of it's **ScenarioEvent**s, and interrelations between them. A scenario event is an instrumented element and its reactive context. One instrumented element in the same reactive context can find itself several times in a scenario and each time it is a different **ScenarioEvent** instance. For example, **getItem** invocations in Figure 2 are specified with two **ScenarioEvent** instances.

Interrelations form a tree composed of **ScenarioEvent**, **Alternatives**, **Contain**, **Precede**, **Negation** and **SubScenario** metaclasses. A transformational context starts with root invocation. A root can be either an instance of **ScenarioEvent** for the scenario containing only one event, or an instance of ***Contain***

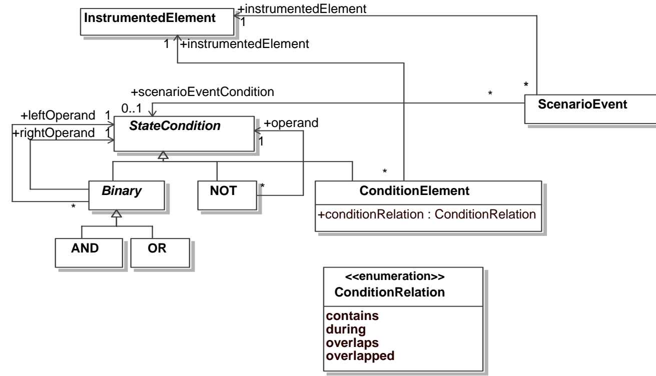

Fig. 4. The part of the metamodel for reactive context specification.

for more complex scenarios. Metaclasses **Contain** and **Precede** enable specification of invocation and sequential composition, respectively. The metaclass **Alternatives** supports specification of guarded command sets. Simple commands are being specified with **ScenarioEvent**. Composite command specified with the usage of **SubScenario** and all other metaclasses mentioned in this paragraph. Guarded commands specification is made possible with the **Negation** and **Precede**, as explained on the example in Subsection 3.1. Finally, loop can be considered as a statement block and it can be specified either as an instrumented element or a composite command.

The reactive context's specification is enabled with the metamodel part shown in Figure 4.

Reactive context is specified with a boolean algebra of active states during the scenario event execution. Furthermore, the interrelation of active states and the scenario event is also taken into account. The boolean algebra is specified with **StateCondition**, **Binary**, **AND**, **OR**, **NOT** metaclasses, and **ConditionElement** metaclass. The possible interrelations are specified in the enumeration **ConditionRelation** enumeration. Based on the assumptions/limitations of the approach, explained in Section 7, and on the ontology of the interval interrelations identified in [8], four possible interrelations are identified: **contains**, **during**, **overlaps**, and **overlapped**. **Contains** is the interrelation between a state and a scenario event where a state starts before and ends after the execution of the scenario event. **Overlaps** is the interrelation in which a state starts before the start of the scenario event execution, but ends before the end of the scenario event execution. **During** and **overlapped** are inverse to **contains** and **overlaps**, respectively.

The **MeasuredEvent** metaclass is used after the context specification for the definition of an event of interest. It contains a context in the ***eventScenario*** attribute, and the event of interest in **measuredScenarioEvent** attribute. Finally, in some cases there is a need for treating several events as one. For example,

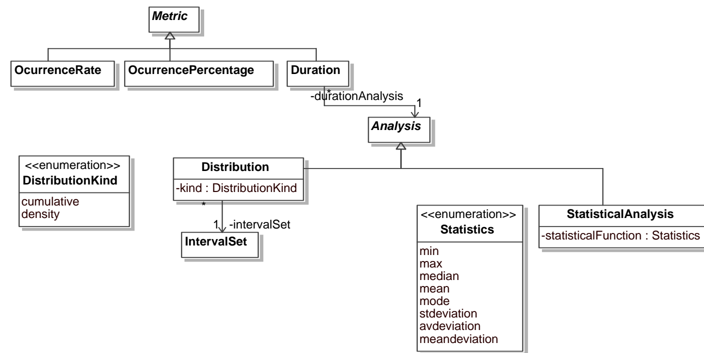

Fig. 5. The part of the metamodel for performance metrics specification.

if a performance analyst would like to measure throughput of a component, he would have to group all methods of that component, and then specify computation of throughput. The **Group** metaclass facilitates grouping of events for which metrics are computed.

The specification of events of interest is followed by specification of desired metrics and time intervals for which they are computed. The metrics metamodel part facilitating metrics specification is presented in Figure 5.

Metrics for performance assessment defined in this metamodel correspond to performance definition in Section 1, and UML SPT [9] and MARTE [10] standard metrics. **Duration** and **OccurrenceRate** metaclasses correspond to response time and throughput, respectively. **OccurrencePercentage** is used for verification of execution probabilities of different alternatives in branching.

Duration of a program construct is being characterized with some statistical functions. Those statistical functions are generalized with the **Analysis** metaclass. Statistical functions are divided into two groups. One group are **distribution functions**, **cumulative** and **density**, defined with instances of **Distribution** and **IntervalSet** metaclasses. Distribution functions are computed as histograms and **IntervalSet** instance defines withs of bars in histograms. The second group of functions are statistical functions which summarize a set of durations in one value. Such metrics' computation is being defined with **StatisticalAnalysis** metaclass instances. Examples of these metrics are mean, median, standard deviation, skewness and so on, and they are defined in the **Statistics** enumeration. This set can be extended. The only requirement is that each function in this enumeration has the corresponding function in the target RDBMS.

Values of all metrics vary over the time. For example, during the peek periods of day response time is higher than in the rest of the day. For this reason, the assessment has to address issues of varying performance metrics values. This is facilitated with the metamodel part in Figure 6.

SimpleAssessment metaclass enables separation of performance assessment time intervals into sub intervals. For example, let the assessment be for

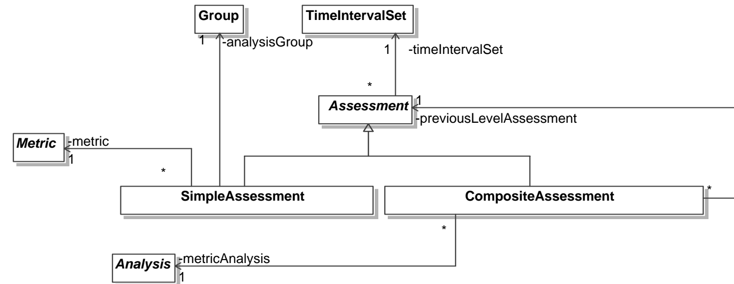

Fig. 6. The part of the metamodel for specification of time varying metrics observations.

a time interval of one day and the metric of interest mean duration. With a **SimpleAssessment** instance and an instance of **TimeIntervalSet** it can be specified that mean duration is computed for each hour of the day. The **TimeIntervalSet** instance defines subintervals for which the metric is computed, here each hour of a day.

CompositeAssessment metaclass enables further statistical analysis of the simple assessment values. For example, with **CompositeAssessment** it can be specified a computation of density distribution of previously mentioned one hour mean durations. Furthermore, for example the standard deviation of one hour mean durations for six hours time intervals can be computed. The time subintervals for composite assessment are also specified with **TimeIntervalSet** instances.

5 Evaluation

The approach is evaluated with an implementation of a UML Profile, and transformations from the profile to Java with RMI and MySQL RDBMS. The UML profile is entitled **PeMA**: The UML Profile for **Pe**rformance **M**easurement and **A**ssessment, and it is, at the present moment, suited only for UML Class and State diagrams. The implementaion in MagicDraw 15.1 Community Edition can be seen in Figure 7.

For these two diagram types the only measurement elements which can be instrumented are operations in Class diagrams and states in State diagrams. The rest of the metamodel is implemented as a model library.

UML Class and State diagrams are transformed into client-server Java RMI applications. For denotation of UML classes modeling client functionality is defined a stereotype ≪Client≫. A corresponding Java class is generated for each class with the ≪Client≫ stereotype. Furthermore, generated are proxies of server classes whose methods are directly invoked by clients.

Server classes are classes without the ≪Client≫ stereotype. For each server class are generated a corresponding Java functionality implementation class and

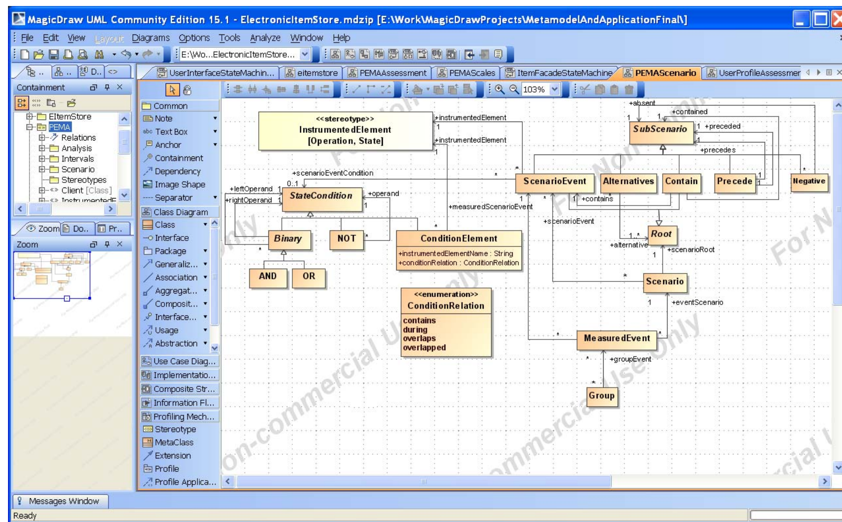

Fig. 7. The implementation UML Profile for **Pe**rformance **M**easurement and **A**ssessment in MagicDraw 15.1 CE. Figure shows the context specification part.

it's instances pool class. Pool classes facilitate concurrent execution defined in Section 7. When a client connects to the server immediately are allocated from pools instances of Java functionality classes to serve to the client. Dispatching between clients and corresponding instances is performed by generated RMI server object class. State charts at the client and server side are implemented with State pattern [11]. These transformations are out of this paper's scope.

Used RDBMS for performance data storage and metrics computation is MySQL 5.2. JDBC MySQL Connector/J driver version 5.1.5 was used as a database driver for performance data storage. Transformations from measurement and assessment part to SQL code for initialization and metrics computation are out of scope of this paper.

Experiments on measuring the duration of the performance data storage procedure were conducted to depict the impact of the measurements to the overall performance. The application was running on the Intel Pentium 4 3.00 GHZ hyperthreaded processor (two virtual cores), 1GB of physical memory, and GNU/Linux 2.6.17.13. The observed value in the experiment was the duration of the performance measurement and data storage routine. Furthermore, it is analyzed with different number of concurrent service requests. For each number of concurrent requests, the experiment was repeated 10 times. Each repetition contained the complete restart of the server, in order to approximate the impact of the distribution of server software over working memory pages.

The experiment was conducted to show the central tendency of the duration of the routine. This should serve as orientation to the performance analyst of how long might the routine last. For this reason was computed the median of the

Concurrent requests	1	10	20	30	60	100
mean(median)	192ms	204 ms	229ms	260ms	289ms	327ms

Table 1. The mean of the median for various concurrent invocations.

duration routine. Then, in order to approximate the value of the data collection and storage routine median, it is computed the mean for all 10 repetitions. The results can be seen in Table 5.

The results show that the performance data collection and storage routine increases with the number of concurrent service requests. In order to obtain the right values of the response times the resulting values from Table 5 for the appropriate number of concurrent invocations should be multiplied by the number of the measurement points at one service and subtracted from the complete measured service response time.

6 Related Work

The measurement and assessment of performance is an important topic in software engineering. This section compares **MoDePeMART**with approaches for performance measurement and assessment at the platforms level, shows their shortcomings, and explains improvements which **MoDePeMART**adds. Subsection 6.1 explains the concerns in performance measurement and assessment and Subection 6.2 shows the comparative analysis of addressing the concerns.

6.1 Comparative Analysis Criteria

One of the major concerns is facilitating statistical analysis of response time. Different kinds of system require different statistical analysis. Furthermore, the parallel analysis of response time and throughput is also needed for validation of meeting SLAs with different number of users. Moreover, workload characteristics observations are important for validation of correspondence of prediction assumptions with test cases. Workload is described with the number, request rate, and arrival pattern.

Characteristics of paths are also of significant interest in measurement and assessment. Path characteristics, such as probability of execution and number of iterations are used in performance predictions.

Not all business tasks are of the same importance in systems, and the most important have to be met in any conditions. Ability of their isolation is of great importance. Furthermore, identification of execution context for critical business tasks is as important as identification of critical tasks themselves.

Performance analysis of software systems has to be done for representative time periods. For example, mean response time of whole day usage must not be the same as during the peek usage period.

Instrumentation transparency is also of great importance in measurement. Additional code for measurement can make the code for business logic more complex and hard to understand. Furthermore, reduction of measurement points is

Approach \ Measurement and Assessment Concern	Response time statistical analysis	Throughput	Workload characteristics (number of requests, request rate, pattern)	Path characteristics (probability in branching, loop iteration numbers)	Isolation of critical business tasks	Specification of execution context (transformational and reactive)	Metrics validity period specification	Instrumentation transparency	Measurement and metric computation data types consistency	Measurement points reduction
Klar et al. [12]	+	-	---	--	o	o-	-	+	+	+
Liao and Cohen [13]	+	-	+--	o-	o	o-	-	+	+	+
Hollingsworth et al. [14]	+	+	++-	o-	o	o-	-	+	+	+
The Open Group [15]	-	-	---	--	o	+-	-	-	+	-
Marenholz et al. [16]	-	-	---	--	o	o-	-	+	-	+
Debusman and Geihs [17]	-	-	---	--	o	o-	-	+	+	+
Diaconescu et al. [18]	+	+	---	--	o	--	-	+	+	-
MoDePeMART	+	+	++-	+-	+	+	+	+(o)	+(o)	+

Fig. 8. Comparative analysis of related work ((+) facilitated, (-) not facilitated, (o) partially facilitated)

also one of the major concerns. It reduces measurement induced system overhead and saves space and time in metrics computation.

Finally, for avoiding assessment failures, keeping consistency between the data structures of collected data and for analysis is also of significant importance.

6.2 The Comparative Analysis

The results of the comparative analysis can be seen in Table 8.

Klar et al. [12] introduced the idea of relating design models and instrumentation. Their approach enabled statistical analysis of durations. The instrumentation is done at the model level, and instrumentation and metrics computation automatically generated. However, there is no possibility of throughput and workload characteristics assessment. Furthermore, there is no negation of an occurrence. For this reason transformational context specification and isolation of business critical task is only partially supported. Reactive context and specification of metrics computation for various intervals is also not supported.

Liao and Cohen [13] and Hollingsworth et al. [14] introduced languages for performance assessment and monitoring. The major shortcomings of these languages are: lack of the reactive context analysis and inability to specify metrics computation for time intervals. Furthermore, due to the lack of the sequence not execution construct the business task isolation, and transformational context specification are only partially supported. Finally, Liao and Cohen [13] do not enable throughput assessment. Application Response Measurement (ARM) standard is an attempt of standardization of data types in performance analysis. This standard addresses the questions of transformation context specification and the consistency of data in measurements and metrics computation. Aspect orientation, on the other hand, e.g. Marenholz et al. [16] solves only the problems of transparent instrumentation. Debusman and Geihs [17] combine AOP and ARM.

Diaconescu et al. [18] add a transparent software layer between components and middleware. Instrumentation is done at component interface, which is not sufficient for context and critical business instrumentation.

MoDePeMARTapproach manages all of the previous mentioned concern except for number of loop iterations analysis. Workload arrival pattern recognition is still not supported.

7 Limitations

MoDePeMARTcan be used in software systems satisfying the following assumptions.

Measurement and assessment is possible only in systems with concurrency without intercommunication. In the execution model it is assumed that there are no concurrent executions which interfere. Moreover, the or invoker of a scenario is not aware of concurrent execution. Such approach is implemented in, for example, JEE Session Beans.

Synchronous communication. At the present time **MoDePeMART**supports only performance measurement and assessment for the systems communicating synchronously. Synchronous communication is the one where the caller of an operation is blocked and waits until the callee returns a result. After the caller gets the result it continues the execution [19].

There is no support for specification of measurement and metrics computation of loopbacks. A loopback is when in a scenario execution control flow reenters the method whose body already executes. The simplest loopback is recursion.

Granularity of timing mechanism is large enough so that execution of each instrumented element occurs in different chronon. Chronon is the smallest unit of time supported by the discrete time model. The granularity is defined with the smallest time units supported by the timing mechanism, such as milliseconds or nanoseconds. The assumption of this approach is that each instrumented element execution with the same sequence identifier executes in different chronon.

Job flow is assumed in the composite occurrence rate assessment. The system should be fast enough to handle the service requests, and thus the competition rate equals the arrival rate.

Finally, **the approach can be used only for verifying response time and throughput of services**. Verifying the equivalence between assumptions on workload, data, and loop iteration numbers in predictions and measurements and in execution is not facilitated.

8 Outlook and Future Work

This paper presents the **MoDePeMART**an approach for model driven performance measurement and assessment. This approach introduces an idea of raising

the abstraction level of measurement and assessment in two ways. First, measurement and assessment is specified in the terms of modeling and not in the terms of implementation constructs. Second, it suggests a DSML for metrics specification and computation. Moreover, it suggests usage of relational database management systems for performance metrics storage and computation. The metamodel for the performance measurement and assessment DSML and a validation as a UML Profile are presented in this paper. With the comparative analysis it is shown that the major benefits of this approach are specification of performance metrics interval computation and the isolation of critical business tasks. However, there are several possible improvements of the metamodel.

The metamodel could be extended in several ways. It could be extended to support performance measurement and assessment of asynchronous communication. Furthermore, the metamodel could be extended to support measurement and assessment of resources utilization. Moreover, the characterization of data used as parameters in services could also be added to the metamodel. Additionally, computation of iteration loop numbers could also be added. This is often needed when assessing the service characteristics. Finally, workload patterns are of great importance for service performance assessment. Extension of the metamodel for workload patterns assessment would be of great usefulness to performance analyst.

Current **PeMA** profile used only State and Class diagrams and both of them are not suited for specification of measurement context. It could be explored usage of activity and sequence diagrams for specification of execution scenario of interest. These diagrams are usually used for control flow description/prescription. This qualifies them as a good basis for transformational context specification. However, still remains the problem of finding the appropriate elements for state context, metrics, and the assessment part of the metamodel. Furthermore, application of the profile to other diagrams could be explored. In extending the profile for application to other diagrams the major challenge is the development of the stereotypes denotating instrumented elements. For example, in the UML metamodel body of activity diagram ***ConditionalNode*** is specified as an attribute. For this reason, it can not be directly annotated as an instrumented element.

The **MoDePeMART**currently facilitates only assessment of services performance. However, it offers a several promising extension directions. With previously mentioned metamodel extension, it could be made very useful in performance debugging or even continuous monitoring. Such language could be support for specification of automatic system adaptation based on the captured runtime performance characteristics.

References

1. Hasselbring, W., Reussner, R.: Toward Trustworthy Software Systems. IEEE Computer, 39(4), pp. 91-92 (2006)
2. I. Sommerville. Software Engineering (8th Ed.). Pearson Addison Wesley, 2007.

3. Smith, C. U., Williams, L. G.: Performance Solutions: A Practical Guide to Creating Responsive, Scalable Software. Addison-Wesley, MA, USA (2001)
4. Balsamo, S., Di Marco, A., Inverardi, P., Simeoni, M.: Model-Based Performance Prediction in Software Development: A Survey. IEEE Transactions on Software Engineering, 30(5), pp. 295-310 (2004)
5. Selic, B.: A Short Course on MDA Specifications, INFWEST Seminar on Model Driven Software Engineering. Pirkkala, Tampere, Finland (2006)
6. Wieringa, R. J.: Design Methods for Reactive Systems: Yourdon, Statemate, and the UML. Morgan Kaufmann Publishers, San Fransisco, CA, USA (2003)
7. Dijkstra, E. W.: A Discipline of Programming. Prentice Hall PTR, Englewood Cliffs, NJ, USA (1976)
8. Allen, J. F.: Maintaining Knowledge About Temporal Intervals. Communications of ACM, 26(11), pp. 832-843 (1983)
9. Object Management Group. UML Profile for Schedulability, Performance, and Time Specification, OMG document formal/05-01-02, http://www.omg.org/cgibin/apps/doc?formal/05-01-02.pdf, January 2005a, Accessed May 2009.
10. Object Management Group. A UML Profile for MARTE: Modeling and Analyzing Real-Time and Embedded Systems, Beta 2, OMG Adopted Spec., OMG document ptc/2008-06-09, http://www.omgmarte.org/Documents/Specifications/08-06-09.pdf, June 2008, Accessed May 2009.
11. Gamma, E., Helm, R., Johnson, R., Vlissides, J.: Design Patterns: Elements of Reusable Object-oriented Software. Addison-Wesley, Boston, MA, USA (1995)
12. Klar, R., Quick, A., Soetz, F.: Tools for a Model-driven Instrumentation for Monitoring. In: the 5th Int'l. Conf. on Modeling Techniques and Tools for Comp. Perf. Evaluation, pp. 165-180. Elsevier Science Publisher B.V. (1991)
13. Liao, Y., Cohen, D.: A Specificational Approach to High Level Program Monitoring and Measuring. IEEE Trans. on Soft. Engineering, 18(11), pp. 969-978 (1992)
14. Hollingsworth, J. K., Niam, O., Miller, B. P., Xu, Z., Goncalves, M. J. R., Zheng, L.: MDL: A Language and a Compiler for Dynamic Program Instrumentation. In Proc. of the 1997 Int. Con. on Parallel Architectures and Compiler Techniques, pp 201-213, IEEE Computer Society, Washington, DC, USA (1997)
15. The Open Group. Application Response Measurement (ARM), http://www.opengroup.org/tech/management/arm, 1998. Technical Standard, Version 2, Issue 4.1, Accessed May 2009.
16. Mahrenholz, D., Spinczyk, O., Schroeder-Preikschat, W.: Program Instrumentation for Debugging and Monitoring with AspectC++. In Proc. of the 5th IEEE Int. Symp. on Object-Oriented Real-Time Distributed Computing, pp 249-256, IEEE Computer Society, Washington, DC, USA (2002)
17. Debusmann, M., Geihs, K.: 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), LCNS vol 2867, pp. 209-220, Springer, Heidelberg, Germany (2003)
18. Diaconescu, A., Mos, A., Murphey, J.: Automatic Performance Management in Component Based Systems. In: 1st International Conference on Autonomic Computing (ICAC'04), pp. 214-221, IEEE Computer Society, Washington, DC, USA (2004)
19. Object Management Group. UML 2.0 Specification: Superstructure, OMG document ptc/05- 07-04. http://www.omg.org/cgi-bin/doc?formal/05-07-04, November 2004. Accessed May 2009.