Model-based Empirical Performance Evaluation based on Relational Traces *

Marko Bošković
marko.boskovic@informatik.uni-oldenburg.de
TrustSoft Graduate School
Carl von Ossietzky University, Oldenburg

Abstract

Empirical performance evaluation is the process of measuring and calculating performance metrics of deployed software systems. It is a part of performance validation during testing of a software system. The topic of this thesis is an approach for the empirical performance evaluation in the context of model-driven engineering. The hypothesis which will be examined is whether concepts of the temporal databases theory can be used as a general way for empirical performance evaluation of model-driven developed software.

1 Introduction

Since the introduction of software engineering, raising complexity of software systems is a persistent problem. High complexity of software makes development harder, and may lead to an increased number of failures. To solve this problem, several attempts like structured, object-oriented and component-based software engineering were introduced. However, the problem still persists. Model-driven engineering (MDE) [10] is an approach which deals with the software complexity by making software models primary artifacts of software development.

A model is a set of statements about the system under study [25]. There are several benefits of this kind of approach but the most important is that a model is much closer to the problem domain then to the underlying implementation. It moves the focus of software engineering from technology specific implementation to the problem domain.

One approach for MDE is Model Driven Architecture (MDA) [16]. The MDA suggests a definition of non-

proprietary standards which define interoperable technologies for the realization of model-driven engineering [20]. It also suggests the usage of the Meta-Object Facility (MOF) [17] for the specification of modeling languages.

To provide trustworthy software, quality attributes [7] have to be satisfied. Performance is a quality attribute which expresses the degree to which a software system or its component meet its objectives for timeliness [27].

Currently, in model-driven engineering most of the research is dedicated to performance prediction with mathematical analysis methods, or with simulation [1][19]. Nevertheless, predictions have to be validated when the software system is implemented and deployed. A validation should be based on modeling constructs as predictions are made. However, the timing behavior is currently observed in the terms of source code constructs (e.g., method execution time).

In MDE the level of abstraction is raised, and models are the primary artifacts of software development. In this case, timing behavior observation of software based on source code constructs is not appropriate. Observations should be done based on modeling constructs (e.g., states, activities, methods or domain specific constructs).

This thesis introduces a general approach for empirical performance evaluation in model-driven developed software. The approach can be applied to any modeling language used for software development. It is based on the temporal databases theory [33]. The temporal databases theory relates facts stored in a relational manner with time information. For this reason, the approach presented here suggests the empirical evaluation based on relational traces. A program trace is a dynamic list of events generated by the program as it executes [14]. In relational traces data about identifying an event and timing information about it are stored in sets of tuples, or tables, in terms of databases. The
The contribution of the thesis is a definition of a basic package for defining instrumentation languages. In software engineering, instrumentation is the process of adding software probes to a program [27]. Software probes are additional pieces of code for collecting data about software execution. From the basic package, a language for instrumentation of some particular modeling language can be derived. Derived instrumentation languages enforce data collection in a relational manner. As examples, two instrumentation languages will be derived, for the UML class diagram modeling language and for the UML state machine modeling language. Furthermore, a trace analysis query language will be developed. This analysis language will provide ability to calculate performance metrics.

The paper is structured as follows. The next section gives an overview of model-driven engineering and the Model Driven Architecture initiative. The motivation and foundations of the approach are presented in Section 3. Section 4 outlines work packages in the research plan. Related work is discussed in Section 5, and Section 6 concludes the paper.

2 Model-Driven Engineering

Modeling is very important for engineering disciplines such as electrical or civil engineering. Engineers use models to express some statements about a system that already exist or that shall be made.

A model is a set of statements about some system under study [25]. Generally, two kinds of models exist: models of existing systems or models of systems which are to be implemented.

Models as a description of existing systems are used to deduct new statements about those systems, or predict behavior of the system in some particular situation. For example, this kind of models are used in physics or chemistry.

As a specification for systems that shall be developed, models are used in engineering disciplines, such as electrical engineering. In this case, models are used to predict characteristics of the system before it is finally produced.

The general idea of MDE is to introduce the model as a first class entity, as it is in the case of other engineering disciplines. With models, software development gets more closer the problem domain, and not to the implementation. They enable the decomposition of problems in a way which is suited to the nature of the system, or part of it, which we are specifying. Generally, there are two kinds of systems, reactive and transformational [31]:

- Reactive systems—systems which are highly interactive with the environment. This kind of systems are able to react to the environment change of states, and enforce certain desirable behavior. Examples of this kind of systems are control, real-time and embedded systems. Information systems as well are reactive systems, because they interact with the organization in their environment. The behavior of this kind of systems is specified with a stimulus/response behavior, such as UML state machines [18].

- Transformational systems—systems which exist to transform an input into an output. Routines in mathematical libraries, compilers, assemblers, and queries over databases are examples of transformational systems. These systems are used only for computation, and when the computation is over, the system is terminated. They can interact with the environment, but only for the purpose of calculation of the output, and not for managing a state of it. Specification of this kind of systems is done with techniques for functional decomposition. With functional decomposition, the system functionality is mapped to the internal components of a system and their collaboration, such as UML class and sequence diagrams [18].

Systems can consist of both, transformational and reactive parts. For example, in the Model-View-Controller [6] three-tier architecture, the Controller is a reactive part, because it defines the way the system reacts to user input. Nevertheless, the generation of Views, from Model is the transformational part, because it generates output from the user input.

Beside moving software development to the problem domain, model driven engineering impedes the ability to exploit formal mathematical methods. Furthermore, with abstraction, understandability can be improved, and state explosion problems in applied mathematical methods can be avoided. Finally, because of the nature of software artifacts, a complete implementation can be generated without discontinuities in the expertise, materials, tools or methods [26]. This implies that the primary and the final artifact of software development in the MDE approach is a model.

The Model Driven Architecture (MDA) [16] is one approach for MDE initiated by the Object Management Group (OMG), a consortium of software vendors and users from industry, academia, and government. The MDA initiative consists of three complementary ideas [2]:

method will be applied in the context of the MDA approach for MDE.
1. **Direct representation**—Shift the focus of software development away from the technology toward the ideas and concepts of the problem domain.

2. **Automation**—Mechanization of relating semantic concepts of a problem domain and an implementation domain.

3. **Open standards**—Usage of non-proprietary standards that will specify interoperable technologies. These standards will close the semantic gap between domain problems and standard implementation technologies.

The topic of this thesis is to enable an empirical evaluation of software performance when the primary artifact of software development is a model. As the approach for model driven development, the MDA will be used. That imposes the usage of open standards defined by the OMG, MOF [17] and UML [18]. In the next section, the motivation and foundations are presented in more detail.

### 3 Foundations and Motivation

#### 3.1 Software Performance, Evaluation and Motivation

Performance is in the thesis considered as the degree to which a software system or component meets its objectives for timeliness [27]. It can be evaluated by simulation, analytical modeling or empirically [14]:

- **Simulation** is a program which enables an imitation of a program execution. In the simulation, only important parts of an execution are imitated. It is less expensive and more flexible than building a real system and then empirically evaluating. However, simulation is not as accurate as measuring real systems.

- **Analytical modeling** is a technique where a system is mathematically described. Results of an analytical model tend to be much less accurate than real system measurements. However, it is the fastest way to get some initial insight into the behavior of the system, or part of it.

- **Empirical evaluation** is performed by measurements and metrics calculation. This provides the most accurate results, as no abstractions are made.

Currently, there exist approaches for analytical evaluation of software performance from annotated models and for simulation [1]. However, there is still not an approach for empirical evaluation of software performance. The topic of this thesis is to introduce an approach for performance evaluation based on modeling constructs.

A necessary part of empirical performance evaluation is the software execution data collection which is achieved by instrumentation. The next subsection gives an overview of instrumentation.

#### 3.2 Instrumentation

In software engineering, instrumentation is the process of adding software probes to the program [27]. Software probes are additional pieces of code for collecting data about the software execution. They can be developed for different techniques of the software execution data collection. Generally, there are two techniques for collecting data about a program execution, sampling and event tracing:

- **Sampling** is a technique where parts of a program are sampled during its execution in some time interval. It is a general statistical technique in which a representative sample of data about the program during execution is taken. In this approach, performance overhead due to measurement is constant. However, a general shortcoming is that the data collected, when the same experiment is performed twice, will hardly be the same. Moreover, infrequent events can easily be missed. An example of this kind of approach is sampling the program stack to follow the execution of a program.

- **Event tracing** is a process of generating traces of events in the software. A program trace is a dynamic list of events generated by the program as it executes [14]. It contains events for characterization the overall program behavior ordered by time. With this technique the processing overhead can be significant in case of very frequent events. Another problem is the size of an event trace. Generally, a trace can be very large if we trace each instruction execution or very frequent events. Large traces occupy resources like memory and, therefore, impact the overall performance of the program.

In the thesis event tracing, as a technique for performance evaluation, will be used. Furthermore, traces will be collected in a relational manner using the concepts of temporal database theory.
3.3 Temporal Databases

Temporal databases are databases which support some notion of time [33]. Conventional (non temporal) databases in which are stored only facts can provide only a possibility of representing a state of a system. For each fact stored in a temporal database a piece of time information.

Facts in databases are represented as rows in tables, or tuples as defined in relational algebra. These facts can be related to the valid time dimension and to the transaction time dimension [33]. 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 [33]. Historical databases define two kinds of relations, event and interval relations [24]. 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 point in time. Those facts were valid only at that time point.

One hypothesis the approach is that all parts of an execution of a program defined by modeling elements of a particular language can be characterized as one of the two previously mentioned kinds of facts.

3.4 Problem, Hypothesis and Conceptual Structure of the Solution

The problem and challenge of this thesis is to raise the level on which performance evaluation is done. Currently, evaluations are performed at the implementation technology level, and mostly based on code constructs. In model-driven developed software, observations of behavior should be done in terms of modeling constructs. Instrumentation, for observing software in the terms of modeling constructs and on the code level would require knowledge of the transformation. For example, if a system is modeled with UML state machines, and assume that implementation is in Java. In that case, instrumentation for observation of timing behavior in the terms of states would require knowledge of the implementation strategy of statecharts. Despite the fact that this is not according to the idea of model-driven engineering, it can be error-prone. Therefore, instrumentation should be done at the model level.

According to the nature of a part of a system, transformational or reactive, appropriate formalisms are used for modeling, as explained in the previous section. Therefore, a system specification uses several modeling languages. The challenge is to introduce the evaluation an approach suited for different modeling formalisms.

The hypothesis of this thesis is that concepts of the temporal database theory can be applied for a performance evaluation approach in the context of model-driven engineering, for all modeling language. With the usage of temporal database theory, we can use already existing implementations of databases for various analyses. Furthermore, we assume that this easily leads to a declarative calculation of performance metrics. As examples for stimulus/response and functional decomposition languages, UML state machines and class diagrams will be used.

The conceptual structure of the solution is presented in Figure 1.

The basic instrumentation package will give the basic structure of traces and the definition of temporal data which can be collected, based on the temporal database theory. The most important concepts in this basic package are interval trace and event trace.

In the case when some element of a modeling language models a part of an execution of the program which lasts for some time interval, it will be instrumented by specialization of the interval trace. Specializations of event traces are used for an instrumentation of parts of executions which only occur at some point in time and do not have a duration. The basic trace types are specialized for each modeling language separately, depending on the type of modeling constructs and the time characterisation of their semantics. For example, in the case of UML class diagrams, there would be only one trace kind, for instrumentation of method calls, and it would be derived from the interval trace. However, in the case of UML state machines there will be event traces for instrumentation of state machines. An event trace will be specialized for instrumentation of outgoing and incoming signals, and for instrumentation of the states, the interval trace will be specialized. The initial structure of the basic instrumentation package is out of scope of this paper.

The trace analysis query language will be also developed. Foundations for this language will be temporal query languages and performance metrics, and it will be general for any instrumentation language.

4 Research Plan

The research plan is divided into the following five work packages:

1. The basic instrumentation metamodel package development

The outcome of this work package is a MOF-based basic package for instrumentation. During this
work package the metrics for performance evaluation will be defined, and the basic instrumentation package will be developed according to them. The methodology which will be used is a survey. The survey analyses existing literature. This work package is currently in progress.

2. **An instrumentation language for a UML class diagrams modeling language development**
   The outcome of this work package is an instrumentation language and a transformation for a UML class diagram based modeling language. In this work package, an MOF based abstract syntax of the language, a concrete syntax in the form of a UML profile, and transformations from a model to Java code will be developed. This work package is in an early phase.

3. **An instrumentation language for a UML state machine modeling language development**
   The outcome of this work package is an instrumentation language and a transformation for a state machine based modeling language. This work package, as the previous, will provide an MOF based abstract syntax, a concrete syntax in the form of a UML profile, and transformations from model to Java code. This work package is in an early phase.

4. **The traces analysis language development**
   The outcome of this work package is a query language for trace analysis. As in the previous cases, the abstract syntax as an MOF defined metamodel. A concrete syntax will text based, and transformations from model to code for analysing traces will be developed.

5. **Feasibility evaluation**
   For a feasibility evaluation, the ArcStyler ([http://www.interactive-objects.com/products/arcstyler](http://www.interactive-objects.com/products/arcstyler)) will be used. In the evaluation, instrumentation languages for Accessors, web tier components for three tier architectures will be developed, and for Components, business object models. Furthermore, transformations from model to code will be developed. Finally, impact of measurements on performance of original code will be done. The feasibility evaluation will be done using the JEE “Pet Store” ([http://java.sun.com/developer/releases/petstore/](http://java.sun.com/developer/releases/petstore/)) case study and the JBoss ([http://www.jboss.org](http://www.jboss.org)) application server.

5. **Related Work**

   At the moment there are several approaches for analytical evaluation of software performance from annotated models [19][23] and for simulation [21]. A detailed survey related to performance prediction can be found in Balsamo et al. [1]. However, there are several approaches for measurement and instrumentation in the context of code-centric development.

   Snodgrass [29] introduces a relational approach for monitoring systems. His work showed that relational
data structures can be an appropriate formalism for monitoring dynamic behavior of a system. According to parts of a system, relations are specified. Furthermore, TQuel [28] is introduced, a language for querying the system state and filtering relational based event traces. In Snodgrass’s approach, the programmer manually defines the instrumentation according to concepts of an existing system. Our approach provides a schema for defining instrumentation languages according to modeling formalisms used for the specification of programs. Furthermore, TQuel provides queries which enable only observation of temporal relations between stored facts. Queries in the query language for performance analysis will enable a calculation of a performance metric through an extension with aggregate functions over temporal data.

Liao et al. [13] introduce a high-level language for program instrumentation and monitoring. In their language, a programmer specifies monitoring and measuring questions. According to the specification, the static analysis of code is done and instrumentation code is added. Conceptually the approach uses a relational approach for the representation of data collected during a program execution. Nevertheless, the language which is developed is only suited for functional decomposition of systems. The approach which is the topic of this thesis shows that temporal relations can be used as a general approach independent of formalism used for decomposition of a software system.

One more language for program instrumentation is the Metric Description Language (MDL), introduced by Hollingsworth et al. [8] as a part of the Paradyn Parallel Performance Tool. The MDL has the ability to define instrumentation as a separate concern, independent of the functionality of a program, define points at which actions for measurement should take place and intertwine it with the program during runtime. Points where measurements can take place are procedure entry, procedure exit and individual call statements. Therefore, as in the previous case the language is suited for functional decomposition only.

The Application Response Measurement (ARM) [30] is a technical standard for response times and for a provision of status of transactions in business applications. It provides a technology independent specification for measurements and implementation can be done in different languages. This standard specifies only a framework for performing measurements, but does not define how to analyse collected data. The methodology in the thesis defines both, how to perform measurements and how to analyze collected data.

Aspect Oriented Programming (AOP) is a programming approach which can be used for transparent software instrumentation. Transparent means that the source code of the software functionality is not mixed with probes. Examples for the application of AOP for an instrumentation are introduced by Marenholtz et al. [15] and Debusmann et al. [3]. Debusmann et al. [3] combine AOP with the ARM standard. Marenholz et al. [15] use AspectC++ for an instrumentation of an operating system for debugging, profiling/measurement, and runtime surveillance/monitoring.

For a transparent instrumentation of CORBA-based component systems, interceptors can be used. Interceptors are similar to AOP and can intercept method invocations to transparently instrument the program. Examples of the usage of portable interceptors in instrumentation are presented by several authors. For example, Li [12] introduces an approach where software probes are predefined and placed in stubs and skeletons during an IDL compilation. The measurement probes can be turned on and off at runtime. Probes are inserted in a program with a modified compiler, which is another technique for transparent instrumentation. Debusmann et al. [4] combine an ARM library and portable interceptors for software instrumentation.

Instrumentation can also be done by adding a transparent software layer to the application. Diaconescu et al. [5] introduce an approach where, at deployment time, a transparent proxy layer for the collection of execution data is automatically generated.

ProbeMeister [22] is a tool for runtime Java bytecode instrumentation. It has an extensible set of software probes that can be inserted or removed at any point while an application is running. In the approach introduced by this tool, the Java Debugging Interface is used for runtime instrumentation. ProbeMeister is a separate application and can be used for instrumentation of multiple JVM’s.

For collecting software execution data, the underlying platform can be instrumented. This kind of approach is presented by Yeung et al. [32]. The authors developed a virtual JVM called Vermeer which is a Java program running on the top of the original JVM. The virtual JVM is a software layer which intercepts class loading, and fragments methods. Each method can be fragmented at the point of a method call, a method entry, basic blocks, and return. After the fragmentation, probes are added to these fragments. Another way to instrument the program is by using a pre-instrumented platform. A pre-instrumented platform is a platform with integrated instrumentation, like Jinsight [9].

Aspect Oriented Programming, portable interceptors, addition of a transparent software layer, bytecode
instrumentation, and underlying platform instrumentation are approaches that enable a collection of data about software execution in a way that the basic functionality code is not influenced. The approach presented in the thesis enables a collection of data about the software execution on a model level, and as a separate concern. Furthermore, it is a performance evaluation approach and not only an instrumentation approach.

Klar et al. [11] developed a set of tools for model-driven instrumentation. They define different sets of models for a program, a functional program model, a functional implementation model, a performance model and a monitoring model. In a functional program model, they explicitly model the functional interdependence of activities without implementation details. A functional implementation model is a detailed model which is concerned with the implementation of the functional program model. A performance model supports validation and is a prerequisite for predicting the performance of not yet implemented programs. In this model, realistic time attributes should be assigned according to measurements performed. Finally, a monitoring model is a subset of the functional implementation model, with which measurement points are defined. Measurements are performed according to the chosen level of abstraction. The source code is instrumented according to this monitoring model. For an analysis tool, a trace description is generated. With a generated trace description, an evaluation tool can decode an event trace. In the approach presented in this paper, the primary artifact of software development is a model. Therefore, instrumentation is done on the model level. The functional implementation model is actually a product of software development. Furthermore, instrumentation defines what to measure and from these two models, source code is generated.

6 Conclusion

Empirical performance evaluation enables validation of timeliness of a software system. However, so far there exist no approach for empirical performance evaluation in the software development process where a model is the primary software artifact.

The paper outlines a research with the purpose to give a general approach for empirical performance evaluation of model-driven developed software. This kind of software development is called model-driven engineering. Instrumentation and empirical performance evaluation at the moment are done based on implementation language constructs. Instrumentation on the implementation technology level for collecting data about a program execution in the terms of modeling constructs can be error prone and can require significant effort. Therefore, the instrumentation is done at the model level. The models for software functionality definition and instrumentation definition are separated to reduce the complexity of models.

The contributions of the thesis are a basic package for the definition of instrumentation languages of UML based models, a methodology for deriving instrumentation languages, and a query language for performance metrics calculation. The languages for instrumentation enable data collection in terms of the modeling language constructs, and structured in relational manner.

Evaluation will be based on the Eclipse modeling tool. UML state machines and class diagram modeling languages, will be used as examples of different decomposition techniques. According to the literature survey on temporal databases and performance, and during development of the instrumentation languages for these two languages, the structure of the basic instrumentation package will be developed. Development of the basic instrumentation package will be followed by the development of a query language for trace analysis.

The feasibility of the approach will be done by integrating software performance evaluation based on relational traces in the ArcStyler commercial tool for model-driven development. Furthermore, experiments on the JEE “PetStore” case study will be performed to show the impact of instrumentation code and execution on the instrumented application.

References


