INSTITUT FÜR INFORMATIK

MAMBA: A Measurement Architecture for Model-Based Analysis

Sören Frey, André van Hoorn, Reiner Jung,
Wilhelm Hasselbring, and Benjamin Kiel

Bericht Nr. 1112
Dezember 2011

CHRISTIAN-ALBRECHTS-UNIVERSITÄT

ZU KIEL

MAMBA: A Measurement Architecture for Model-Based Analysis

Sören Frey, André van Hoorn, Reiner Jung, Wilhelm Hasselbring, and Benjamin Kiel

Software Engineering Group, University of Kiel, D-24098 Kiel, Germany
{sfr, avh, rju, wha, bes}@informatik.uni-kiel.de

***Abstract*—Model-based measurement techniques are relevant in the field of software analysis. Several meta models for the specification of quantitative measures have been proposed. However, they often focus either on static or dynamic aspects of a software system. Nevertheless, considering reengineering activities often both dimensions reveal valuable complementary insights. Existing meta models are also frequently bound to specific modeling languages, redefine underlying concepts for any new meta model, or provide only limited tool support for the automated computation of measurements from modeled measures.**

We present MAMBA, an integrated measurement architecture for model-based analysis—both static and dynamic—of software systems, that can be specified by arbitrary Ecore-based modeling languages. MAMBA extends the Structured Metrics Meta-Model (SMM) by additional modeling features, such as arbitrary statistical aggregate functions and periodic aggregate functions, e.g., for dynamic analysis at runtime. To consider measurements for querying system models, we outline the MAMBA Query Language (MQL) that employs SMM measures. Furthermore, we provide tool support that applies the measures specified in an (extended) SMM model and can integrate raw measurements provided by arbitrary static and dynamic analysis tools to produce the desired measurement model.

We demonstrate the applicability of the approach based on three evaluation scenarios from different contexts: migration of software systems into the cloud, model-based engineering of railway control systems, and dynamic analysis for model-driven software modernization.

I. Introduction

Software analysis spans a wide range of methodologies, domains, and techniques, such as model checking, code analysis, and dependency extraction [1]. For example, a system's behavior can be observed at runtime to collect information needed for program comprehension [2] or architecture measures can be computed to reason about a software product's quality [3]. The empirical analysis of software models that comply with well-defined meta models can be seen as a sub-discipline of software analysis. Here, as in any empirical field of research, being able to measure properties that are related to the subjects of interest is vital. Several approaches for carrying out model-based measurement have been proposed (e.g., [4–6]). However, the corresponding meta models often include domain-specific elements, focus on either dynamic or static aspects of software systems, or underlying concepts are redefined for each new meta model. Furthermore, the tool support is frequently limited and the measures can often only be used in conceptual models but cannot be computed automatically. Using measures in model queries is also often cumbersome as those require customized full-fledged measure models. Moreover, it is often not feasible to reuse already measured values that are provided by arbitrary static and dynamic analysis tools. Those externally computed raw measurement data usually cannot be smoothly integrated in new measurement models.

In this paper, we describe our measurement architecture for model-based analysis (MAMBA) that facilitates the specification and automated application of measure models upon arbitrary Ecore-based modeling languages. It utilizes OMG's Structured Metrics Meta-Model (SMM) [7] and builds upon the Metrics Execution Engine (MEE) that we briefly introduced in our previous work [8]. MAMBA constitutes a substantial reworking of MEE and forms a reusable and extensible measurement framework.[1] It addresses the shortcomings of the current approaches that are sketched above through contributing significant extensions:

(1) We add modeling features that are missing in the current V. 1.0 Beta 3 of SMM. Here, we include features such as arbitrary statistical aggregate functions and periodic aggregate functions that can also be used for dynamic analysis at runtime. (2) We outline the MAMBA Query Language (MQL) that enables the utilization of SMM measures for querying software models. MAMBA transparently transforms MQL statements and the therein referenced measures to SMM instances that are computed implicitly. (3) The MAMBA framework provides means to smoothly integrate externally computed raw measurement results. Corresponding hooks enable to plug in appropriate model to model transformations.

We report on three evaluation scenarios that are used to evaluate our approach. Here, we demonstrate the applicability of MAMBA in the context of the following different domains: migration of software systems into the cloud, model-based engineering of railway control systems, as well as dynamic analysis for model-driven software modernization.

The remainder of this paper is structured as follows. We give an overview of SMM in Section II to outline the foundations of our work. The MAMBA approach including MQL is then described in Section III. The three evaluation scenarios that demonstrate the applicability of MAMBA are presented in Section IV. Section V discusses the related work, before Section VI concludes the paper and outlines future work.

[1] http://mamba-framework.sf.net/, visited December 20, 2011

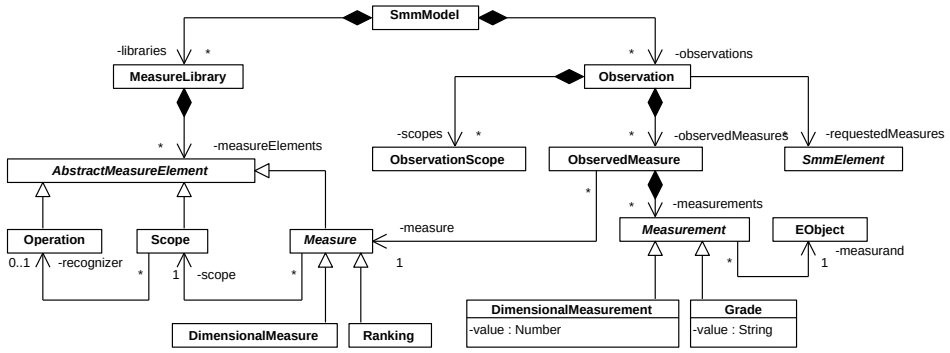

Fig. 1. Core classes of the SMM meta model (using the EObject class for *measurand*s instead of a MofElement)

II. Structured Metrics Meta-Model (SMM)

The *Structured Metrics Meta-Model (SMM)* [7] is one of the meta model specifications developed by the *Architecture-Driven Modernization (ADM)* Task Force,[2] a sub-commitee of the Object Management Group (OMG). SMM provides generic means to specify elements relevant for the domain of model-based measurement. In the remainder of this section, we give an overview on SMM and elaborate on several of its shortcomings we addressed in the course of our work building MAMBA.

A. Overview

In the context of model-based reconstruction and evolution of software systems, the ADM Task Force develops MOF-based meta models which allow to represent software systems and related artifacts on different layers of abstraction regardless of concrete technologies, e.g., programming languages or platforms used. Complementing ADM specifications in addition to SMM, so-called standards packages, are the *Knowledge Discovery Metamodel (KDM)*[3]—OMG's foundation for software modernization, recently adopted as an ISO standard—and the *Abstract Syntax Tree Metamodel (ASTM)*.[4]

The SMM standard employs the central notion of *measures* to describe methods for computing values upon MOF-based models. The term is used as a synonym for *metric* that is widespread as well. Hence, a measure describes an algorithm for calculating specific properties of a software system's elements. A software function's cyclomatic complexity, the contained lines of code, or its average response time constitute classic examples of such measures. The meta model defined by SMM enables to specify those measures—along with the computed results and further concepts specific for the measurement domain—following an abstract representation that facilitates its utilization as a common interchange format that can be used by different tool vendors. Now, we will describe the basic ideas and elements of SMM through referring to Fig. 1 that shows the meta model's core classes, as, for example, the central class Measure.

[2]http://adm.omg.org/, visited December 20, 2011
[3]http://www.omg.org/spec/KDM/, visited December 20, 2011
[4]http://www.omg.org/spec/ASTM/, visited December 20, 2011

A Measure can be applied to a specific set of model elements that is defined through the Scope class. For example, a Scope may limit the computation of the cyclomatic complexity to source code snippets and therefore exclude BLOB files that can be present in extracted system models. In addition, to consider only a subset of the source files (e.g., those containing C source code) an Operation could be formulated that specifies this restriction using OCL or XQuery. The classes of SMM mentioned so far inherit from the AbstractMeasureElement class. Corresponding child classes can get registered and be supplied via a MeasureLibrary. Furthermore, Fig. 1 illustrates the two concrete measures Ranking and DimensionalMeasure. The first may, for example, classify a method as *low prio* or *high prio* depending on its cyclomatic complexity residing in $[0, 40)$ or $[40, \infty)$, respectively. The DimensionalMeasure describes a measure that assigns a numeric value. The counterpart of a Measure is a corresponding Measurement that holds the result which is produced through executing the Measure. The counterparts of the previously mentioned classes Ranking and DimensionalMeasure are the Measurements Grade and DimensionalMeasurement. A model element that was measured is referenced via the *measurand* relationship of the respective Measurement. This could be a specific method model element, for instance. As we want to measure Ecore-based models, *measurand* points to an EObject in the context of MAMBA. A concrete measurement process is encapsulated via an instance of Observation and can therefore be distinguished from other measurement runs. The Observation contains information such as the time of the measurement and it describes the actually used Measures and the measured elements through referencing ObservedMeasure and ObservationScope, respectively.

Further measures that are relevant for describing our SMM extensions and evaluation scenarios in the subsequent sections are presented in Fig. 2. A DirectMeasure can measure a model element through applying an Operation. Counting is a specific DirectMeasure. It is used to restrict its Scope to a relevant subset of model elements as the referenced Operation returns 0 or 1 for a given *measurand*. A NamedMeasure denotes a familiar measure that can be described unambiguously by solely stating its name. BinaryMeasures apply two measures—referenced via Base1MeasureRelationship and

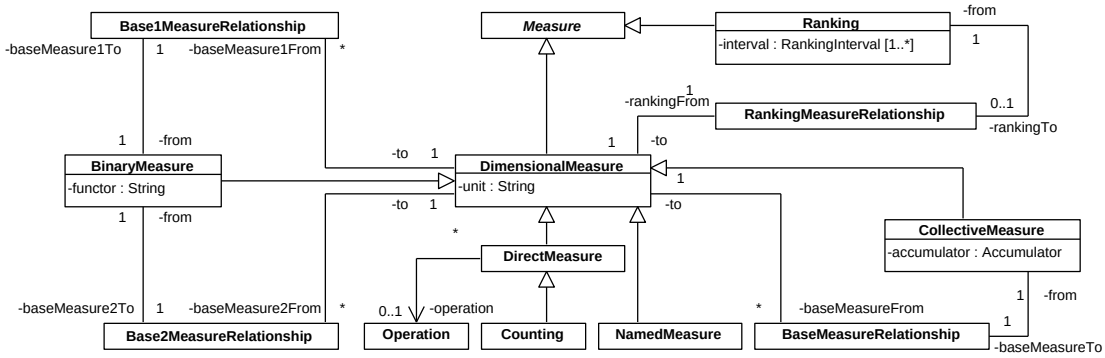

Fig. 2. SMM's Measure classes

TABLE I
COMPARISON OF PURE SMM AND MAMBA

Issue	**Pure SMM**	**MAMBA**
Meta models	MOF	Ecore
Model execution	Limited tool support	MEE
Raw measurement data integration	No tool support	Via measurement providers
Model querying using measurements	No native support	MQL: implicit calculation and integration of measurements in queries
Periodic measures	No native support	Periodic collective MAMBA measures
SMM as runtime model	No native support	Continuous execution with MEE
Aggregate functions	*Limited set*: sum, maximum, minimum, average, standard deviation	*Extensible set*: sum, maximum, minimum, average, standard deviation, median, percentile etc.

Base2MeasureRelationship—to a model element and then evaluate a binary function upon the corresponding measurements, for example, to calculate their difference. A CollectiveMeasure enables to apply an accumulator to any number of collected base measurements, e.g., to compute the standard deviation.

B. Shortcomings

The SMM specification contains the CollectiveMeasure class modeling the accumulation of measurements for an associated base measure (DimensionalMeasure) into a single value. However, using the Accumulator enumeration, the SMM specification limits the set of supported aggregate functions to *sum*, *maximum*, *minimum*, *average*, and *standard deviation*. This way, common aggregate functions, such as median or other percentile functions, cannot be used with SMM so far.

Moreover, especially in service-level management, collective measures are applied to bounded sets of contiguous base measurements. For example, in SLA documents the definition of QoS measures, e.g., addressing availability and performance, is based on time periods. The main reason is that short-term QoS degradations are hidden by aggregations over long-term periods. The corresponding aggregate function is then applied in periodic time steps, considering base measurements

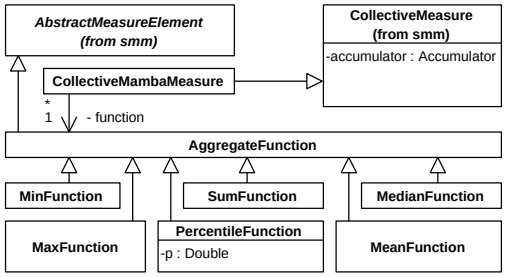

Fig. 3. MAMBA extension mechanism for collective measures

observed during the elapsed time period of specified length. Currently, the SMM specification has no support for modeling periodic measures of any kind. A further challenge when adopting SMM's current V. 1.0 Beta 3 can be seen in the lack of the specification's maturity. Although the basic structure and ideas of SMM are encouraging, there still exist some inconsistencies that currently impede its interoperability and that should be addressed.

With our MAMBA approach presented in this paper, we aim to address some of the SMM's current shortcomings, as detailed in the following Section III. Table I provides a compact comparison of pure SMM and MAMBA.

III. MAMBA APPROACH

The following sections describe (III-A) our meta model extensions for arbitrary aggregate functions and (periodic) collective measures; (III-B) the MAMBA Execution Engine (MEE); and (III-C) the MAMBA Query Language (MQL). Note that in our previous work [8], MEE was used as an abbreviation for *Metrics* Execution Engine. The work presented in this paper builds upon our previous work and we decided to revise the meaning of the abbreviation.

A. Meta Model Extensions to SMM

a) Arbitrary Aggregate Functions: In order to make the set of supported aggregate functions extensible, we added a new meta model class AggregateFunction (abstract) into our SMM extension, as shown in Fig. 3. Also included are six example aggregate functions. The parameterized PercentileFunc-

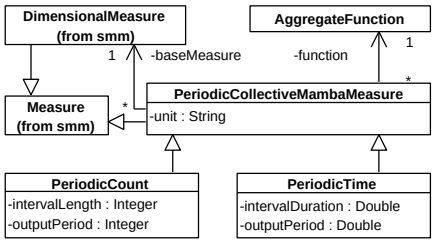

Fig. 4. MAMBA extension mechanism for periodic measures

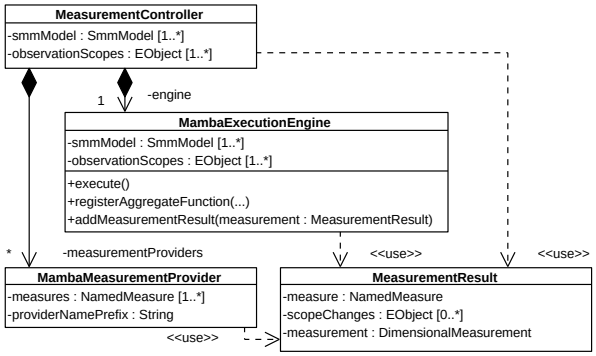

Fig. 5. Core MEE components and their relationships

tion class demonstrates the limitation of using enumerations or strings to select aggregate functions.

MAMBA users can now use these or custom aggregate functions in (periodic) collective measures, as detailed in the following paragraphs. Custom functions can be defined by using meta model classes that extend AggregateFunction.

b) Extended Collective Measures: Since SMM's CollectiveMeasures cannot use the newly introduced AggregateFunctions, we added another meta model class CollectiveMambaMeasure. CollectiveMambaMeasure extends the SMM class CollectiveMeasure and references an AggregateFunction, introduced above. Fig. 3 shows the meta model extensions regarding the newly introduced collective measure and associated aggregate function.

c) Periodic Collective Measures: We included support for modeling periodic measures in our SMM extension by introducing the abstract meta model class PeriodicCollectiveMambaMeasure. Just like the CollectiveMambaMeasure class described above, it references a MAMBA aggregate function (AggregateFunction). This meta model extension is depicted in Fig. 4. PeriodicCollectiveMambaMeasure extends Measure rather than DimensionalMeasure (or CollectiveMambaMeasure) in order to preclude semantic ambiguity, for instance because the latter would allow PeriodicCollectiveMambaMeasure to be used as base measure of collective measures. Currently, two concrete classes for periodic collective measures are included in our SMM meta model extension (see also Fig. 4): PeriodicTime and PeriodicCount. PeriodicTime can be used to model measures where the referenced aggregate function is repetitively applied with a time period (*outputPeriod*) incorporating the measurements observed within the elapsed time period of length *intervalDuration*. PeriodicCount triggers the aggregate function to be computed for every *outputPeriod*-th new measurements, incorporating the past *intervalLength* measurements.

B. MAMBA Execution Engine (MEE)

This section provides a description of the MAMBA Execution Engine (MEE) with respect to its architecture (III-B1), its execution of open SMM models, as defined below (III-B2), and its extension mechanism for aggregate functions (III-B3).

1) Framework Architecture: The core components in the MEE architecture are a MeasurementController, the actual MambaExecutionEngine, as well as a (potentially empty) set of MambaMeasurementProviders. The UML class diagram in Fig. 5 depicts these components and their relationships.

From given resource URIs, the MeasurementController loads SMM instances along with the Ecore models which constitute the Oberservation's ObservationScope (see Section II-A)—e.g., KDM models to be analyzed—, and inspects the given set of requested Measures. For each measurement run, the MeasurementController creates an Oberservation which is passed to an instance of the MambaExecutionEngine responsible for the execution of the SMM model, i.e., computing the Measurements for the Observation with respect to the requested Measures.

We distinguish between the execution of *closed* and *open* SMM models. A closed SMM model contains no NamedMeasures and MEE can execute these models directly, without requiring any additional input. Details on the execution of closed models with MEE have been presented earlier [8].

For *open* SMM models, the Measurements corresponding to NamedMeasures are provided by so-called MambaMeasurementProviders. On instantiation—given an *open* SMM model—the MeasurementController looks up appropriate MambaMeasurementProviders by name prefixes matching among NamedMeasures (fully qualified *name* attribute) and the MambaMeasurementProvider's *providerNamePrefix*.

2) Execution of Open SMM Models: MambaMeasurementProviders integrate with external tools for static and/or dynamic analysis by importing raw measurement data for the supported NamedMeasurements from the tools' output, and transforming this raw data into MeasurementResult objects which are delivered to the MeasurementController. Each of these MeasurementResult objects (see Fig. 5) contains information on the observed NamedMeasure and a DimensionalMeasurement. The MeasurementController delegates these MeasurementResult objects to the MambaExecutionEngine which appropriately incorporates the measurement into the SMM model.

In continuous scenarios, an ObservationScope may evolve during the measurement process, e.g., adding components when discovering software architectures from incoming monitoring data. MambaMeasurementProviders indicate such changes by setting the *scopeChanges* in a MeasurementRe-

sult object. In this case, the MambaExecutionEngine needs to reprocess the SMM model before incorporating the new measurement.

3) Extension Mechanism for Aggregate Functions: As described in Section III-A, our SMM meta model extension allows the use of custom aggregate functions in SMM instances using meta model classes extending MAMBA's AggregateFunction. For being able to compute the respective measurements for (periodic) collective measures using custom aggregate functions, MEE requires the registration of appropriate functionality.

For this reason, MEE provides the method *registerAggregateFunction*, requiring a meta model class name of a custom aggregate function (extending AggregateFunction) as well as a class and operation name for a respective function handler. MEE instantiates a singleton instance for each registered handler class. When executing a (periodic) collective MAMBA measure, the registered handler operation is called by MEE with the referenced AggregateFunction instance and the base measurement's values passed as arguments. Listing 1 shows how this mechanism is used internally by MEE to register its own custom, but already included, aggregate function PercentileFunction. Listing 2 shows the handler signature.

```
String smmPercFunctionClass = "mamba.smm.extended.PercentileFunction";
String mambaFunctionsClass = "mamba.mee.handler.MambaFunctions";

this.mee.registerAggregateFunction(smmPercFunctionClass,
      mambaFunctionsClass, "percentileFunction");
```

Listing 1. Registration of handler for PercentileFunction

```
public double percentileFunction(PercentileFunction function, double[] values);
```

Listing 2. Handler signature for PercentileFunction

C. MAMBA Query Language (MQL)

The *MAMBA Query Language* is designed to utilize measures modeled in SMM to query Ecore-based models. The language vocabulary follows the naming scheme of other query languages to ease the learnability. It is divided in clauses that describe the result set, reference the models under observation, and declare constraints on measures (**SELECT** result **FROM** models **WHERE** constraint **GROUP BY** identity **ORDER BY** value **AGGREGATE OVER** period **OUTPUT EVERY** period).

Measures in an SMM model have a *name* attribute. That name is used in MQL to identify a measure and to apply it to the observed model. As a convention, measures visible to MQL, use "Measure" as a suffix to their name. For example, "ResponseTimeMeasure" in an SMM model is called "ResponseTime" in MQL. Furthermore, MQL can express the instantiation of additional measures by specifying the class name of SMM measure classes. Measures can appear in **SELECT**, **WHERE**, **ORDER BY**, and **GROUP BY** clauses. Every specified measure is applied to the observed model.

While a measure and respective measurements in SMM belong to separate class hierarchies, in MQL all properties of a measure and its measurements are expressed by the name of the measure followed by a dot and the name of the property.

In a **WHERE** clause, measures are used in boolean expressions to select measurements. As each Measurement uses the *value* property to store their result value, the *.value*-part can be omitted resulting in better readable expressions (e.g., **WHERE** NrOfWebservices >= 1).

The use of measures in the **SELECT** clause is quite similar to that of the **WHERE** clause. The specification of a measure name represents the value of a measurement for that measure. The measurand can be accessed through the *measurand* property (e.g., NrOfWebservices.*measurand*). Alternatively, the class name (e.g., KDM::CodeModel) of the measurand can be specified to instruct MEE to return the measurand and limit the result to measurands of the specified type.

While the definition of measures can be complex and repetitive in SMM (compare Fig. 9), MQL allows to implicitly add measures to an SMM model based on the class name of the respective measure. For example, the maximum of all results of a measure named ResponseTimeMeasure in the SMM model is computed by a CollectiveMambaMeasure with the AggregateFunction *max* (CollectiveMamba.max(ResponseTime)). The MQL resolver only requires the name of the AggregateFunction (e.g, max(ResponseTime)) to specify a CollectiveMambaMeasure or, if an **AGGREGATE OVER** clause is present, a PeriodicCollectiveMambaMeasure.

Without further specification, the result is an unordered list of records conforming to the result specified in the **SELECT** clause. As known from other query languages like SQL, the result can be sorted by specifying criteria with an **ORDER BY** clause and grouped by a **GROUP BY** clause. Periodic measures are supported by MQL through two language constructs: First, **AGGREGATE OVER** is used to specify the sample window; and second, the output can be triggered periodically with **OUTPUT EVERY** followed by a time expression.

An MQL query can be used in annotations to a model or multiple queries can be stored in one model. For the latter case, it is helpful when the observed model can be specified once and used in multiple queries. This is done with **MODEL** modelURI **AS** modelName, where modelURI references one or more models, and modelName is a valid identifier name.

The model of interest is often scattered across several separated artifacts. Therefore, MQL allows to reference multiple artifacts in a comma-separated list, and to specify model URIs which reference a folder or collection of artifacts and models.

IV. Evaluation Scenarios

This section reports on the application of MAMBA to three different evaluation scenarios from the following domains: migrating software systems into the cloud (Section IV-A), model-based engineering of railway control systems (Section IV-B), as well as dynamic analysis for model-driven software modernization (Section IV-C). For each scenario we provide (1) a brief project description, (2) describe the application of MAMBA in this context, including the manual definition of SMM measures, and (3) provide example MQL queries and its translation into SMM measures.

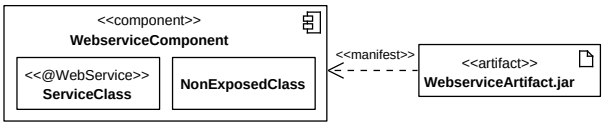

Fig. 6. Components that expose web services may be deployed in separate virtual machines during a migration. Following JSR 181, Java classes can be marked as web services using the annotation "WebService."

A. Migration to the Cloud

1) Context: Cloud computing has recently gained considerable attention as a promising paradigm for delivering software, platforms, and infrastructures as services over a network connection [9]. In the cloud computing context, these service models are abbreviated SaaS, PaaS, and IaaS, respectively. Following the SaaS model, users can consume applications from remote servers. These applications most often can merely be configured to a limited extent. In contrast, PaaS and IaaS cloud environments allow to deploy custom-made software systems. Here, PaaS-based cloud environments provide predefined software stacks whereas IaaS providers rather deliver fundamental IT services such as virtual machines. In [10], we introduced our approach CloudMIG that supports reengineers in migrating existing enterprise software systems to PaaS and IaaS-based clouds. CloudMIG incorporates the reverse engineering of existing systems and for that purpose utilizes extracted KDM models. These models are employed to detect violations of so-called cloud environment constrains (CECs) which can be automatically checked with CloudMIG. For example, those CECs may limit the access to the file system. In [8], we already utilized SMM to detect CEC violations. Furthermore, CloudMIG follows a rule-based approach to facilitate reasoning about cloud deployment options. For example, a rule can describe a strategy to distribute components that expose web services to own virtual machines.

2) Application of MAMBA: Considering such components that expose web services, Fig. 6 shows a component named WebserviceComponent that contains two classes. As indicated by the according stereotype, the class ServiceClass is augmented using a WebService annotation. We use Java in our example and consider the Java Specification Request (JSR) 181. Here, the aforementioned annotation marks a Java class as a web service and by default exposes all public methods. The WebserviceComponent is implemented in the library WebserviceArtifact.jar. Fig. 7 presents the KDM representation of the example scenario. The shown objects are instances of KDM elements. We extract KDM models by producing a CodeModel element for any present Java library. Classes and interfaces are modeled using ClassUnit and InterfaceUnit elements, respectively. Therefore, ServiceClass is represented as a ClassUnit code element that is referenced from a CodeModel named "WebserviceArtifact.jar". The Java general purpose annotation facility uses a notation that is similar to a standard Java interface declaration to construct an annotation type. Thus, the Java annotation type javax.jws.WebService is translated to a KDM InterfaceUnit instance that is declared in

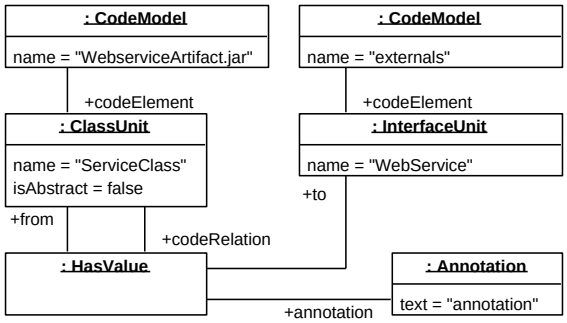

Fig. 7. KDM model of the web service example in the CloudMIG context

an external CodeModel. This InterfaceUnit is connected to the web service's ClassUnit via a HasValue relationship class that is interlinked with an instance of the class Annotation. To mark the KDM annotation as an annotation of the target language —Java in our example—the instance of the class Annotation has "annotation" as the value of its text attribute.

In this example, we want to count the number of JSR 181 web services that are exposed by WebserviceArtifact.jar and therefore analyze the KDM model with MAMBA using the SMM instance presented in Fig. 8a). The measure we are interested in is termed "NrOfWebServicesMeasure", counting the web services that are contained in a KDM CodeModel. Hence, it is modeled using a CollectiveMeasure that sums up every single web service that is found by the Counting measure named "CountWebserviceMeasure." The main part of this measure's logic is implemented in the referenced Operation instance that contains an OCL expression in its *body* attribute. The OCL expression checks if a ClassUnit contains a HasValue relationship via its *codeRelation* role that is specific for the JSR 181 web service annotation. This means that an appropriate Annotation instance exists and that an InterfaceUnit termed "WebService" is referenced via the *to* relationship.

3) MQL in the Cloud Migration Scenario: Applying the previously introduced measure "NrOfWebServicesMeasure" to a KDM model that was extracted from a software system yields the number of exposed JSR 181 web services for every single Java library. To conveniently isolate the names of the libraries that expose at least one JSR 181 web service, we employ this measure in the MQL query that is presented in Fig. 8b). According to its defined scope, the measure "NumberOfWebServices" is executed by MAMBA for any KDM CodeModel that can be found in the folder that is stated in the **FROM** clause. Then, the measurement result is used to evaluate the predicate that is specified in the **WHERE** clause. As stated before, every Java library is described in an own KDM CodeModel in the context of CloudMIG. Therefore, we can distill the name of a matched library by querying the *name* attribute of the corresponding CodeModel element.

B. Model-Based Analysis of Railway Control Centers

1) Context: The development of electronic railway control centers (RCC) is a costly endeavor, often proven unviable for light railways. To divert more traffic to railways and to

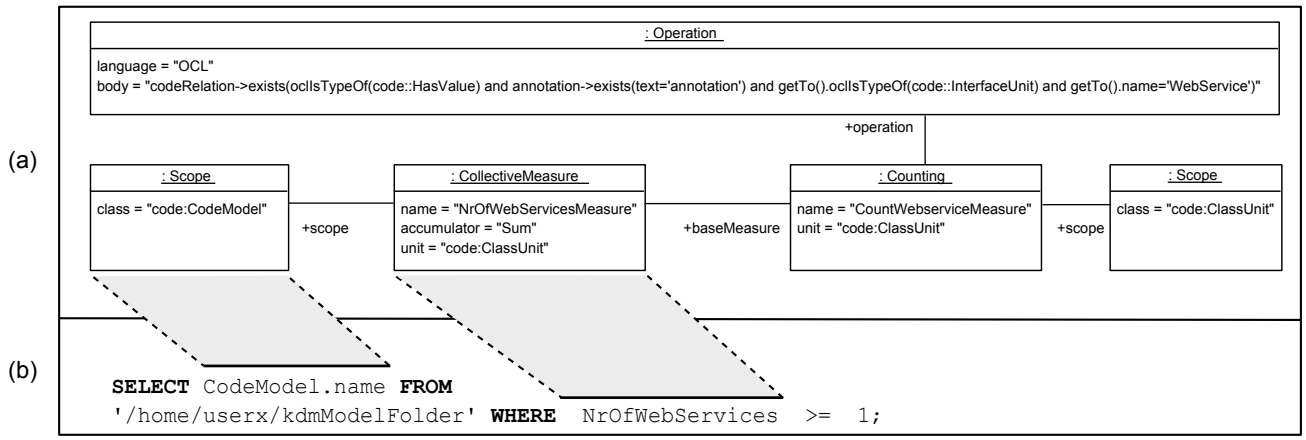

Fig. 8. (a) An SMM instance for counting web services in KDM-based models. (b) An MQL query that uses this SMM instance to select the names of KDM CodeModel elements that include at least one web service. These names represent the names of file system artifacts.

reduce cost for the operation of the infrastructure, the control systems have to be modernized to meet these requirements. Therefore, the hardware components have to become cheaper and the development of the software components has to be more productive without compromising the safety.

Today, process computers are used for electronic RCC, which have to be tailored for each RCC; also the software has to be developed for the respective facility. In our MENGES research project[5] we aim to improve productivity utilizing standard hardware components for industry automation, in this case programmable logic controllers (PLC), and applying model-driven software development methods. To ensure the safety of the developed software, it has to be statically and dynamically evaluated. In MENGES we developed four domain-specific languages (DSLs) which address different aspects of the problem domain. The central DSL, called *types*, is used to declare data types, classes, interfaces, and connectors. The behavior of those types is defined in the *logic* DSL which provides structures to define DFA, actions, conditional actions, and workflows. The instantiation in safety critical systems has to be static and is defined in the *instantiation* DSL. A model defined in these languages is transformed into an execution model which can be executed on a PLC-simulator like CoDeSys[6] or on real hardware.

The evaluation of the software's runtime properties is performed with MAMBA. It is used to define measures, apply corresponding instrumentation to the execution model, and analyze the collected measurements to determine properties like the component response time or to detect unused or seldom used code to guide optimization efforts.

2) Application of MAMBA: In this paper we illustrate the use of MAMBA in MENGES with the analysis of the response time of a switch control component (SCC) to the turn-over command for a railway switch. The response time is an important property of an RCC's components, as they add up to the total response time of the system, which must not exceed a certain duration to meet the safety constraints.

The command and its possible responses are declared in a construct called CommunicationDescription which declares two roles for a communication, a set of messages, and protocols based on these messages. In Listing 3 these roles are CommandInterpreter and SwitchControl, where the first is normally assigned to a command interpreter component and the second is assigned to an SCC. The protocols are declared after a **rules** keyword. The rule in Listing 3 describes the protocol for a turn-over command, where a command interpreter sends a *turnOver* message to an SCC. That SCC then answers either with *accept* or *reject*.

```
communication description SwitchControlCom : CommandInterpreter,
      SwitchControl {
   messages
      turnOver();
      accept();
      reject(Cause cause);
   rules
      CommandInterpreter turnOver ->
      SwitchControl accept|reject -> CommandInterpreter ;
}
```

Listing 3. Declaration of a communication protocol

For the evaluation of the SCC response time to the turn-over command, the maximum and the distribution of response times in relation to the answers (*accept* and *reject*) are of interest. The maximum is important to calculate the overall response time of the RCC for worst case scenarios. The other two values are helpful to support optimization efforts. The response time is modeled with a BinaryMeasure and two NamedMeasure instances which collect the time of sending the *turnOver* command and receiving the *accept* or *reject* answers.

The SMM model in Fig. 9 shows the SwitchControlResponseTimeMeasure and three CollectiveMambaMeasure instances to declare the measuring of the maximum response time, as well as the lower and the upper quartile of the response times. The CollectiveMambaMeasures and required other instances (shown in gray) are automatically added to the SMM model through the MQL queries.

[5]http://kosse-sh.de/projekte/menges/, visited December 20, 2011
[6]http://www.3s-software.com/, visited December 20, 2011

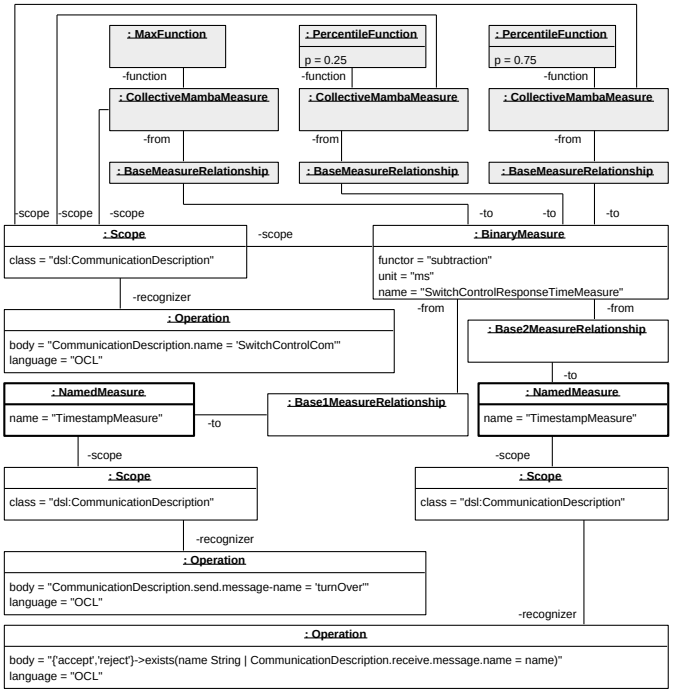

Fig. 9. Measuring response time of a SCC to a *turnOver* command

3) MQL use in a Multi-DSL Scenario: In MENGES, the code generator for the DSL uses an SMM model to generate an instrumented execution model. In this model only the BinaryMeasure and NamedMeasures from the SMM model are used to drive the instrumentation. After the code generation, the execution model is executed on/processed by a simulator and the results are collected and stored in the SMM model.

During or after the simulation run, the SMM model is evaluated with the MQL queries to return the lower and upper quartile as well as maximal response time (see Listing 4).

```
SELECT percentile(SwitchControlResponseTime,0.25) FROM model ;
SELECT percentile(SwitchControlResponseTime,0.75) FROM model ;
SELECT max(SwitchControlResponseTime) FROM model ;
```

Listing 4. Excerpt of the response time MQL model

As only the SwitchControlResponseTimeMeasure is used during the instrumentation of the execution model, the query model can be extended with additional queries as long as these new queries do not introduce new base measures. For example, the average response time can be determined from the same measurements.

C. Dynamic Analysis for Model-Driven Modernization

1) Context: In the DynaMod consortial research project[7] we investigate techniques for model-driven modernization (MDM) of software systems. Similar to the ADM approach, we focus on sustained modernization by incorporating architectural and domain-level concerns rather than restricting us to implementation-level transformations of legacy systems. Key characteristics of the DynaMod approach are: (1) combining static and dynamic analysis for extracting models of a legacy system's architecture and usage profile; (2) augmenting these models with information relevant to subsequent architecture-based modernization steps; and (3) employing model-driven techniques for generating implementation artifacts and test cases based on the information captured in the models.

The DynaMod ADL (DADL) is the core structure of the DynaMod meta model catalog for representing architectural views on both legacy and modernized systems. Relevant to the application of MAMBA in DynaMod, described in the following section, is the DADL meta model partition allowing to represent type-level information about an application. A *type model* contains a number of *component types*, implementing a set of software *operations* (meta model class Operation).

For different programming platforms, DADL instances can be extracted by static and dynamic techniques—including the combination of both. For example for Visual Basic 6 (VB6) applications, a source code parser extracts abstract syntax trees (ASTs); these ASTs are transformed into a VB6 language model for which another transformation into a DADL instance exists. For the extraction of a DADL instance based on dynamic analysis, we employ our Kieker framework for continuous monitoring and analysis of software systems[8] [11]. In the DynaMod context, we added support for monitoring the runtime behavior of VB6 and .NET applications, in addition to the already existing support for Java-based systems. A Kieker plugin allows to extract a DADL instance from the application's observed runtime data and usage profile—including the refinement of a DADL instance extracted by the above-mentioned static technique.

2) Application of MAMBA: This section demonstrates the application of MAMBA for performance analysis of software operations. In this example, we want to study the following performance characteristics for each software operation contained in a DADL instance: (C1) the number of executions of each operation; (C2) whether 95% of the observed executions satisfy an operation response time objective of 500 milliseconds; and (C3) time series with 0.95 response time percentiles for time windows of 20 minutes, computed every 15 minutes for each operation.

Figure 10 shows the SMM model elements relevant for modeling C1–C3. The model includes two DynaMod-specific NamedMeasures, one for operation execution counts and a second for operation response times. C1 is already realized by the NamedMeasure "dynamod::OperationExecutionCountMeasure". The second NamedMeasure "dynamod::OperationExecutionResponseTimeMeasure" serves to model C2 and C3. As both C2 and C3 rely on response time aggregation based on a 0.95 percentile function, we employ the parameterized aggregate function PercentileFunction provided by MAMBA. For C2, this aggregate function is used by the CollectiveMambaMeasure which aggregates the entire set of response time observations, grouped by

[7]http://kosse-sh.de/dynamod/, visited December 20, 2011

[8]http://kieker.sourceforge.net/, visited December 20, 2011

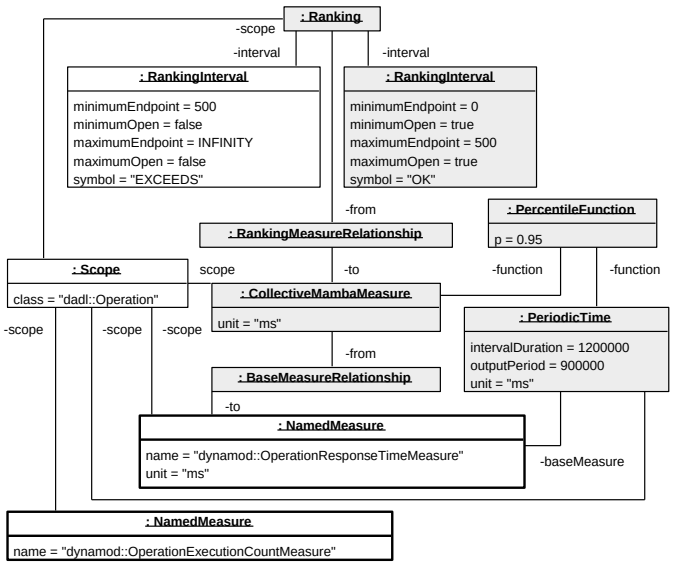

Fig. 10. Example measures for dynamic analysis in the DynaMod context

operations. C3 is already realized by using this function in a MAMBA PeriodicTimeMeasure with appropriately parameterized interval duration and output period attributes. The classification of operations based on their characteristic of exceeding/satisfying the mentioned response time objective is realized employing the Ranking measure with two RankingIntervals. Since we want to have the measurements grouped by operation, each of the measures has the same Scope with the DADL meta model class Operation.

The DimensionalMeasurements for the two NamedMeasures, processed by the MAMBA execution engine, are made available by a custom MAMBA data provider by transforming and pre-processing Kieker monitoring records received for each execution of instrumented operations. Note, that this set-up can be used for offline and online analysis. We employ model transformations to generate the monitoring instrumentation based on the modeled measures [12]. In a model discovery mode, the data provider extends the DADL instance during the analysis based on the incoming monitoring data.

3) MQL in the DynaMod Scenario: Listing 5 shows MQL queries implementing the three performance measures C1–C3 from the previous paragraph.

```
−− C1
SELECT OperationExecutionCount
  FROM Bookstore.dadl GROUP BY Operation

−− C2 (selecting only SLO violations)
SELECT percentile(OperationResponseTime, 0.95) as rtPerc
  FROM Bookstore.dadl WHERE rtPerc > 500 GROUP BY Operation

−− C3
SELECT percentile(OperationResponseTime, 0.95)
  FROM Bookstore.dadl GROUP BY Operation
  AGGREGATE OVER 20 min OUTPUT EVERY 15 min
```

Listing 5. MQL queries related to C1–C3 from the DynaMod context

In each query, the **SELECT** clause refers to one of the two DynaMod-specific NamedMeasures which can be found in the manually created SMM model from Fig. 10. Also, each of the queries uses the **GROUP BY** clause to group the result set by Operations, resulting in the SMM Scope being set appropriately. Due to the **AGGREGATE OVER** clause in the third query, the percentile aggregator in the **SELECT** clause is mapped to a PeriodicTimeMeasure. In addition to the parameterization of the PeriodicTimeMeasure, the **OUTPUT EVERY** clause leads to a result set being sent to the query subscriber every 15 minutes. The gray Measures in Fig. 10 emphasize the parts of the SMM model, that are generated from the MQL queries.

V. Related Work

Related work comes from other methodologies and meta models for model-based measurement that are not built on SMM. Nonetheless, there exist further approaches and tools that use SMM as well.

A. Methodologies for Model-Based Measurement

Another software measurement framework (SMF) is described by Mora et al. [4]. SMF uses an own Software Measurement Metamodel (SMM) that, in spite of the identical abbreviation, is not to be confused with OMG's SMM. To measure a MOF-based software model using SMF, an instance of their SMM has to be created by utilizing the graphical or textual syntax of the Software Modeling Measurement Language (SMML), for instance. The measurement execution is then performed by SMF through automatically applying QVT transformations. Measurements are parameterized through modifying OCL queries. Despite several similarities, MAMBA exhibits a number of unique characteristics: (1) Measures can be incorporated in MQL statements that enable to dynamically query software models based on measurement results that are computed implicitly. (2) MAMBA provides means to integrate externally computed raw measurement results. (3) We consider dynamic analyses through incorporating periodic aggregate functions and demonstrate the applicability in this context.

Within the scope of a software measurement validation framework, Kitchenham et al. [13] define a structural model that describes basic entities that are involved in software measurement. Then, five additional models are proposed that, in contrast to OMG's SMM, separate related concerns like instrumentation and measurement protocols.

A model-driven measurement approach is presented by Monperrus et al. [5]. Measures are formulated according to the so-called metric specification meta model. This meta model and referenced domain meta models utilize EMF. According to the specification of a measure, an Eclipse plugin is generated that performs the measurement process for an instance of a domain meta model.

B. Approaches and Tooling Based on SMM

Engelhardt et al. [6] use SMM to measure arbitrary domain-specific models. At first, the Object Constraint Language (OCL) is used to specify measures. This representation includes generation rules that are applied to create SMM instances. The generated SMM measures can be computed with

their tool Metrino. Comparing the two approaches, MAMBA allows to build measures using directly SMM elements.

SMM is considered by Larrucea and Iturbe [14] to measure MOF-based software process models. The authors describe integration models for combining the Software Process Engineering Metamodel (SPEM) and the jBPM Process Definition Language (JPDL) with SMM. The SMM meta model is extended by a "variable" concept that allows to use different types for measurement results.

VI. Conclusions

We presented the integrated measurement architecture MAMBA for analyzing software system models. It builds on OMG's SMM standard that specifies a meta model for defining a measurement process. For example, SMM describes measures and observations. MAMBA supports the execution of SMM measures upon Ecore-based models and can be utilized for static as well as for dynamic software analyses. The combination of both analysis types into a hybrid one can often provide additional insights, e.g., the detection of unused components that were revealed during an architecture reconstruction step applying data from runtime analyses in the reengineering context. Therefore, we extended SMM to enable the integration of additional aggregate functions and periodic measures that can be applied to runtime models. Furthermore, the extensible MAMBA architecture allows to register data providers for incorporating measurements that were computed by external software tools. The integration of raw measurement data into used measurement models has the potential to raise the level of reuse and to improve the efficiency of projects that apply software analyses. To ease model querying with the use of SMM measures, we introduced our query language MQL. We demonstrated the applicability and versatility of MAMBA, with the help of three evaluation scenarios based on different contexts.

Currently, we are refining MQL and are working on appropriate tool support for a convenient definition, registration, and execution of MQL queries. Also, we will provide a comprehensive SMM library including additional measures relevant in our project domains. Furthermore, we are investigating methods to extend MAMBA to further simplify the integration of additional periodic measures and aggregate functions.

Acknowledgment

This work is supported by the German Federal Ministry of Education and Research (BMBF) under grant number 01IS10051, the Program for the Future Economy of Schleswig-Holstein, and the European Regional Development Fund (ERDF).

References

[1] D. Jackson and M. Rinard, "Software Analysis: A Roadmap," in *Proceedings of the Conference on The Future of Software Engineering*, ser. ICSE '00. ACM, 2000, pp. 133–145.

[2] B. Cornelissen, A. Zaidman, A. van Deursen, L. Moonen, and R. Koschke, "A Systematic Survey of Program Comprehension through Dynamic Analysis," *IEEE Transactions on Software Engineering*, vol. 35, no. 5, pp. 684–702, 2009.

[3] K. M. Hansen, K. Jonasson, and H. Neukirchen, "An Empirical Study of Software Architectures' Effect on Product Quality," *Journal of Systems and Software*, vol. 84, no. 7, pp. 1233–1243, 2011.

[4] B. Mora, F. Garcia, F. Ruiz, and M. Piattini, "Model-Driven Software Measurement Framework: A Case Study," in *9th International Conference on Quality Software (QSIC '09)*, 2009, pp. 239–248.

[5] M. Monperrus, J.-M. Jézéquel, B. Baudry, J. Champeau, and B. Hoeltzener, "Model-Driven Generative Development of Measurement Software," *Software and Systems Modeling*, pp. 1–16, 2010.

[6] M. Engelhardt, C. Hein, T. Ritter, and M. Wagner, "Generation of Formal Model Metrics for MOF based Domain Specific Languages," *ECEASST*, vol. 24, 2009.

[7] Object Management Group, Inc., "Architecture-Driven Modernization (ADM): Structured Metrics Meta-Model (SMM), V. 1.0 Beta 3," http://www.omg.org/spec/SMM/.

[8] S. Frey, W. Hasselbring, and B. Schnoor, "Automatic Conformance Checking for Migrating Software Systems to Cloud Infrastructures and Platforms," *Journal of Software Maintenance and Evolution: Research and Practice*, 2011, (To appear).

[9] M. Armbrust, A. Fox, R. Griffith, A. D. Joseph, R. H. Katz, A. Konwinski, G. Lee, D. A. Patterson, A. Rabkin, I. Stoica, and M. Zaharia, "Above the Clouds: A Berkeley View of Cloud Computing," EECS Dept., Univ. of California, Berkeley, Tech. Rep. UCB/EECS-2009-28, 2009.

[10] S. Frey and W. Hasselbring, "Model-Based Migration of Legacy Software Systems to Scalable and Resource-Efficient Cloud-Based Applications: The CloudMIG Approach," in *Proceedings of the 1st International Conference on Cloud Computing, GRIDs, and Virtualization (Cloud Computing 2010)*, 2010, pp. 155–158.

[11] A. van Hoorn, M. Rohr, W. Hasselbring, J. Waller, J. Ehlers, S. Frey, and D. Kieselhorst, "Continuous Monitoring of Software Services: Design and Application of the Kieker Framework," Dept. Comp. Sc., Univ. Kiel, Germany, Tech. Rep. TR-0921, 2009.

[12] A. van Hoorn, H. Knoche, W. Goerigk, and W. Hasselbring, "Model-Driven Instrumentation for Dynamic Analysis of Legacy Software Systems," in *Proceedings of the 13. Workshop Software-Reengineering (WSR '11)*, 2011, pp. 26–27.

[13] B. Kitchenham, S. Pfleeger, and N. Fenton, "Towards a Framework for Software Measurement Validation," *IEEE Transactions on Software Engineering*, vol. 21, no. 12, pp. 929–944, 1995.

[14] X. Larrucea and E. Iturbe, "A Metamodel Integration for Metrics and Processes Correlation," in *ICSOFT (1)*, 2010, pp. 63–68.