Generic Instrumentation and Monitoring Description for Software Performance Evaluation *

Alexander Wert[1], Henning Schulz[2], Christoph Heger[1], Roozbeh Farahbod[2]
[1]Karlsruhe Institute of Technology, Am Fasanengarten 5, Karlsruhe, Germany
[2]SAP AG, Vincenz-Priessnitz-Strasse 1, Karlsruhe, Germany
alexander.wert@kit.edu, henning.schulz@sap.com, christoph.heger@kit.edu, roozbeh.farahbod@sap.com

ABSTRACT

Instrumentation and monitoring plays an important role in measurement-based performance evaluation of software systems. To this end, a large body of instrumentation and monitoring tools exist which, however, depend on proprietary and programming-language-specific instrumentation languages. Due to the lack of a common instrumentation language, it is difficult and expensive to port per se generic measurement-based performance evaluation approaches among different application contexts. In this work-in-progress paper, we address this issue by introducing a performance-oriented, generic meta-model for application-independent and tool-independent description of instrumentation instructions. Decoupling the instrumentation description from its realization in a concrete application context, by a concrete instrumentation tool allows to design measurement-based performance evaluation approaches in a generic and portable way.

1. INTRODUCTION

Many performance engineering tasks entail measurement-based analysis of the application under test (AUT), such as performance troubleshooting, capacity planning, performance regressions testing, etc [11]. Most of these tasks employ instrumentation and monitoring (IaM) tools to retrieve performance data from the AUT. There is a large body of different IaM tools with different, proprietary and programming-language-specific configuration languages.

Though many measurement-based performance engineering tasks are conceptually transferable from one application context to another, the lack of a common, generic and abstract configuration language for the IaM tools renders portability of established performance engineering processes among different AUTs (potentially based on different technologies) impractical. On the one hand, in many performance engineering projects multiple, different IaM tools are used, requiring expertise in different instrumentation languages and a way of keeping different instrumentation descriptions consistent. Furthermore, introducing a new IaM tool entails tedious migration of existing instrumentation descriptions. On the other hand, without a common instrumentation language, applying conceptually similar instrumentation and monitoring instructions on different contexts (i.e., projects) implies tedious analysis of the target AUT in order to create AUT-specific instrumentation descriptions for each concrete context.

Existing languages for the description of instrumentation instructions [2, 3, 5, 7] either lack abstraction from technical implementation details, are programing-language-specific, or are not designed for measurement-based performance evaluation. As such, they are not sufficient to enable generic description of instrumentation instructions, which can be reused among different application contexts.

In this work-in-progress paper we present the Instrumentation Description Model (IDM) which is a generic meta-model for describing instrumentation and monitoring instructions in an AUT-independent, IaM-tool-independent way for the purpose of software performance engineering. Primarily designed for AUTs implemented in object-oriented, managed programing languages, IDM instances can be reused even in different environments, such as Java or .NET. We show the conceptual relationship between IDM, potential AUTs and existing IaM tools. Furthermore, we discuss the main design goals for IDM and describe the semantics of the individual model elements.

2. INSTRUMENTATION DESCRIPTION

In this section, we explain the context of IDM, discuss the main design goals for an abstract instrumentation description model and show how we realized these goals by explaining IDM in detail. Finally, we show an example of an instantiation of IDM.

2.1 Giving the Context

The main goal of IDM is to decouple the instrumentation and monitoring descriptions from their realization in a concrete managed-language-based context. Hereby, concrete context means both the application under test (AUT) and the instrumentation and monitoring (IaM) tools used to instrument the AUT.

IDM is a meta-model for the specification of instrumentation descriptions (IDM instances). An IDM instance can reference concrete parts of a specific AUT (e.g. concrete methods), generic concepts common in most AUTs (e.g. the

*The research leading to these results has received funding from the DFG grant RE 1674/6-1.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
ICPE'15, Jan. 31–Feb. 4, 2015, Austin, Texas, USA.
Copyright © 2015 ACM 978-1-4503-3248-4/15/01 ...$15.00.
http://dx.doi.org/10.1145/2668930.2695525.

concept of a database interface), or both. An IDM instance is generically reusable among different contexts (AUTs and IaM tools) if it only comprises instrumentation description entities which reference generic concepts without referencing AUT-Specific parts. Specific instrumentation engine are responsible for mapping generic concepts to concrete implementations of the AUT. For instance, in the context of Java, a Java instrumentation engine would map the concept of a database interface to the JDBC API. Hereby, it is irrelevant how a specific instrumentation engine realizes the instrumentation described by an IDM instance, whether through static code analyses, load-time weaving [5] or even dynamically adaptable instrumentation [1]. Existing IaM tools can be adapted to use IDM by providing an adapter which interprets and translates IDM instances to the tool-specific and context-specific specification.

2.2 Design Goals for IDM

In order to address the issues mentioned in Section 1, we identified the following design goals for IDM:

Abstraction. In order to reuse instrumentation descriptions in different contexts, we need a meta-model which allows to abstract from the specifics of individual run-time environments, programming languages and concrete AUTs. However, if an AUT-specific instrumentation is required, the meta-model has to provide means to describe an AUT-specific instrumentation, too.

Orthogonality. An instrumentation instruction has basically two dimensions: *Where* to instrument (in the following called *scope*), and *How* to instrument (which *probes* to inject). Since entities of both dimensions can be comprehensive, definitions of scopes and probes need to be independently reusable. Therefore, as far as reasonable, probes need to be independent of scopes, and vice versa.

Composability. In order to provide a flexible and expressive way of describing instrumentation instructions, the meta-model needs to be composable. Besides the orthogonality of probes and scopes, individual model elements should cover basic, minimalistic aspects of the AUT and the measurement data of interest. In this way, instrumentation descriptions can be kept simple (e.g. in order to keep the measurement overhead low), while advanced descriptions can be composed when needed.

Focus. An instrumentation description language with a clear focus on a domain allows to define expressive models while reducing complexity and effort of model creation. Our focus lies on performance evaluation which guides the definition of probes (i.e. data to measure).

2.3 IDM in Detail

Following the design goals (cf. Section 2.2), we created IDM as depicted in Figure 1. In general, IDM can be partitioned along two dimensions. Hereby, we distinguish between *sampling* and *instrumentation* on the first dimension, and between *scopes* and *probes* (cf. orthogonality principle, Section 2.2) on the second dimension. Sampling is the process of periodically retrieving information from a certain resource (e.g. CPU utilization, Disk I/O, etc.) while instrumentation is used to retrieve measurement data (e.g. response times) from the control flow of the AUT. While scopes describe which resource to monitor, respectively where in the code to instrument, probes define what should be measured.

The `IaM Description` is the root element of IDM, constituting a container for `Instrumentation Entities` and `Sampling Entities` which are explained in the following.

2.3.1 Instrumentation

An `Instrumentation Entity` comprises an `Instrumentation Scope` (i.e. *where* to measure) and a set of `Measurement Probes` (i.e. *what* to measure). We distinguish different types of scopes:

Synchronization Scope. This scope covers all points in the AUT, where synchronization of threads occurs due to a lock on a passive resource. In particular, this scope covers all lock acquisition events and lock release events.

Method Enclosing Scope. This is an abstract scope, providing a common parent element for all scopes which cover a set of individual methods or method calls. Entities of this scope type comprise two parts: a *before-method-execution* and an *after-method-execution* part.

Method Scope. The Method Scope is the most direct way to specify a set of methods to be instrumented. Hereby, the methods are identified by a set of `method patterns` (method names with potential wild-cards). A Method Scope covers all methods (except for constructors) whose full qualified names match a specified pattern.

Constructor Scope. Covers the instrumentation of all constructors of the target classes. Analogously to the method patterns the target classes are specified by name patterns.

Allocation Scope. Covers all code statements where an object of the target classes is allocated. By contrast to the `Constructor Scope`, this scope does not instrument any constructor, but only their invocation.

Modifier Scope. The Modifier Scope allows to specify a set of modifiers (e.g. `public`, `private`, etc.) to describe a scope of all methods whose modifiers match all specified modifiers of the scope.

API Scope. Conceptually, an `API Scope` covers all methods of an *abstract API* whose implementing methods shall be instrumented. An *abstract API* represents a conceptual interface for which concrete APIs consist in most modern, managed programing languages (e.g. Java, .NET, etc.). In the current version, IDM supports the following five API scopes: The `Entry Point API` covers all entry points into a server application (e.g. in Java: Servlets, HTTP handlers, etc.). The `Messaging API` covers all ways of the target context to realize remote communication (e.g. in Java: JMS, RMI, etc.). The `Database API` and the `OR Mapping API` need to be mapped to the concrete standard API of the target context for managing a database connection (e.g. JDBC in Java), respectively concrete API for realizing an object-relation mapping (e.g. JPA in Java). Finally, the `Logging API` covers standard interfaces for logging (e.g. log4j, slf4j).

Trace Scope. The `Trace Scope` allows to instrument the whole dynamic trace (i.e. call tree) originating from the methods covered by the `sub-scope`.

The `Method`, `Allocation` and `Constructor Scope` provide a application-specific way of specifying a scope. Application-specific definition of instrumentation instructions may be useful in some application scenarios where the AUT is know and reuse of the instrumentation descriptions is not important. However, in an evolving environment, where IaM tools may be replaced, or common measurement-based performance evaluation approaches need to be applied on several AUTs,

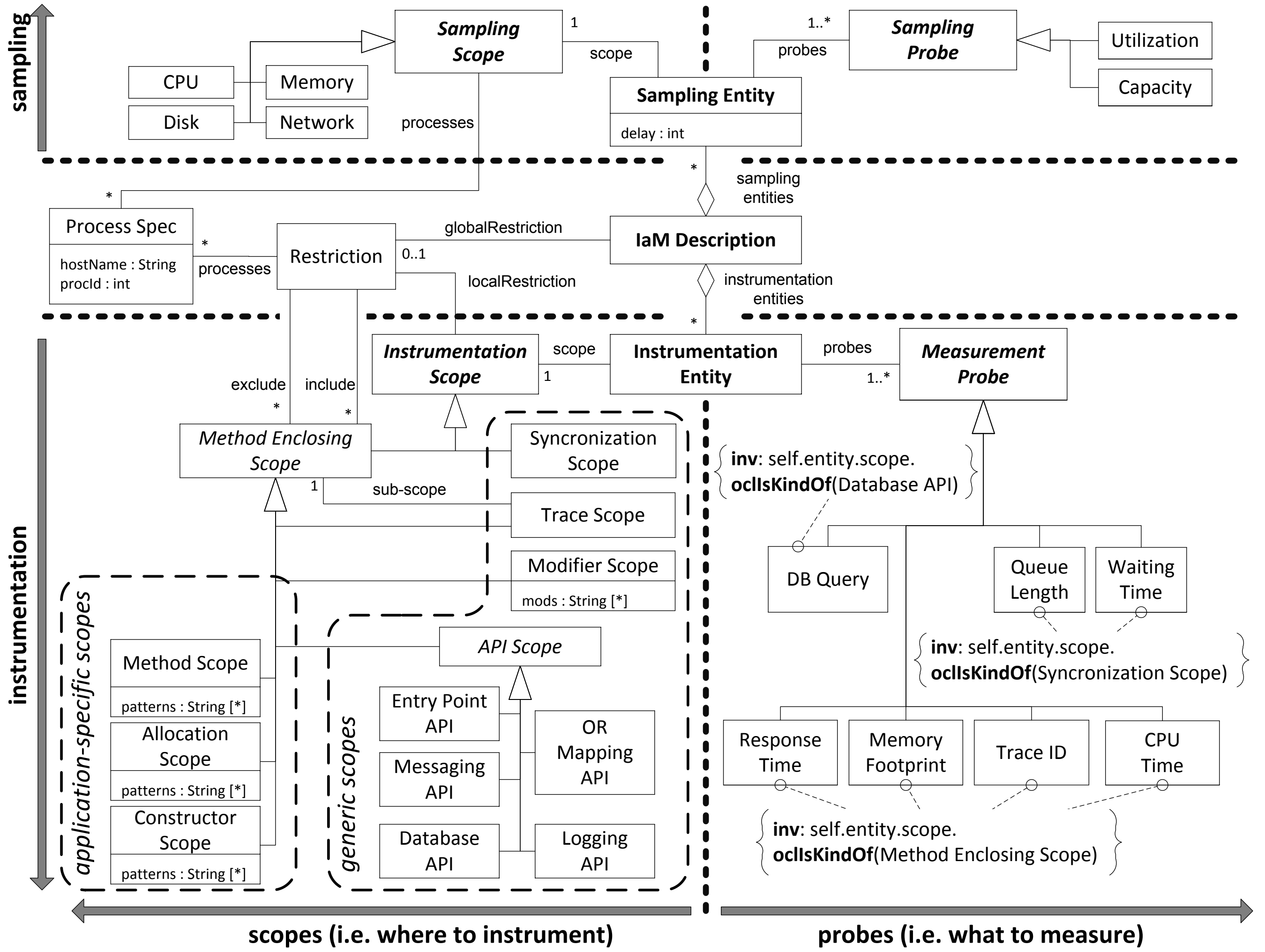

Figure 1: Instrumentation Description Model (IDM)

purely generic instrumentation descriptions are required. To this end, IDM provides the remaining scopes (`Synchronized, Trace, Modifier` and `API Scope`), which allow to describe scopes in an AUT-independent way.

Following the composability principle (cf. Section 2.2), an `Instrumentation Scope` can be further refined by a local `Restriction`. In particular, a `Restriction` allows to limit a scope to a set of processes (specified by a host name and process id) and a set of excluded, respectively included, scopes. Hereby, the restriction has the following semantics: Let us assume that M is the set of all methods in the AUT, S is the set of methods defined by a scope X (without regarding the restrictions), and In_i, Ex_j are the inclusive, respectively exclusive, scopes of the restriction for X. Then, the scope X is resolved to the following set of methods:

$$X = S \cap \left(\bigcap_i In_i \right) \cap \left(\bigcap_j M \backslash Ex_j \right) \tag{1}$$

Additionally to the local restrictions, an `IaM Description` can have a global `Restriction` which applies to all `Instrumentation Entities`.

Orthogonally to the `Instrumentation Scope`, an `Instrumentation Entity` comprises a set of `Measurement Probes` specifying the data to be retrieved from the corresponding scope. Although the probes are orthogonal to the scopes, not all probes are reasonably applicable with all scopes. Therefore, we define some restrictions (expressed as OCL statements) limiting the applicability of certain probes to corresponding sub-sets of scopes. In particular, `Response Time`, `Memory Footprint`, `Trace ID` and `CPU Time` can be measured from all `Method Enclosing Scopes`. `Waiting Time` and `Queue Length` can be retrieved from the `Synchronization Scope`. Finally, the SQL statement of an executed `DB Query` can be retrieved from a `Database API` scope.

2.3.2 *Sampling*

A `Sampling Entity` is specified by a delay (in ms) determining the sampling frequency, exactly one sampling scope (e.g. CPU, Memory, Disk or Network), and a set of sampling probes (e.g. Utilization and Capacity). The `Capacity` probe stands for the absolute capacity of the corresponding resource, e.g. the absolute clock rate of a CPU, the network bandwidth or total memory. The Sampling Scope can be further restricted to a set of operating system processes

(`Process Spec`) specified by a host name and a process id. In this way one could monitor for instance the CPU utilization of a single process instead of the overall CPU utilization.

2.4 IDM Instance Example

Figure 2 shows an exemplary, target application-independent, however, Java-specific IDM instance. On the sampling side of the `IaM Description`, the IDM instance specifies sampling of the utilization of all CPUs with a frequency of 2Hz (cf. `delay=500ms`). On the instrumentation side the goal is

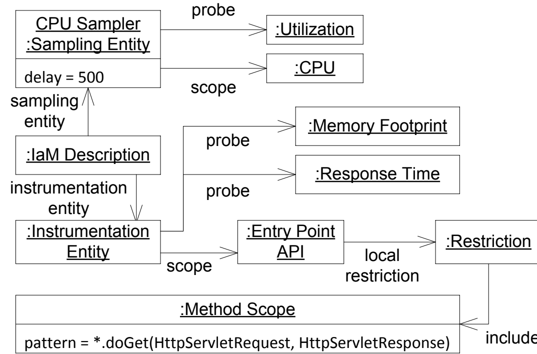

Figure 2: An exemplary IDM instance

to capture the response times and memory footprints (cf. `probes`) of all `doGet(...)` methods (cf. inclusive restriction) of all Java Servlets (covered by the `Entry Point API`). Note, the method pattern `*.doGet(...)` as part of the restriction is the only Java-specific entity in this example.

3. RELATED WORK

As instrumentation is a cross-cutting concern, most instrumentation languages and tools are based on ideas of aspect oriented programming (AOP) [6]. AspectJ [5] is an implementation of AOP in Java comprising its own language for instrumentation definition. AspectJ (similarly to IDM) distinguishes between where to instrument (pointcuts) and how to instrument (advices), which is specified in a Java-like syntax. Josh [3] is an AspectJ-like language introducing a mechanism for defining custom pointcut designators in Java. SCoPE [2] is an AspectJ compiler allowing to provide user-defined analysis-based pointcuts. DiSL [7] is a domain-specific instrumentation language. Relying on concepts of AOP, DiSL provides high level language constructs based on Java and Java annotations to describe instrumentation instructions for an AUT. Providing an open join point model, DiSL is able to instrument any region of Java bytecode, ranging from methods through loops to single statements. All muti-purpose, AOP-based languages [2, 3, 5, 7] inherently lack a focus on performance evaluation, requiring additional definition of probe-code. Furthermore, AOP-based languages are specific to the target programing language they are designed for, rendering general, language-independent reuse of instrumentation descriptions impossible.

Kieker [9] is a framework for continuous monitoring and performance analysis of software utilizing existing AOP solutions for instrumentation. Although Kieker does not comprise its own instrumentation language, it provides AOP advices for measurement-based performance evaluation simplifying the definition of performance-oriented instrumentation descriptions. Though Kieker supports different programing languages, descriptions of instrumentation instructions are programing-language-specific.

The Java Performance Measurement Framework (JPMF) [8] introduces a generic interface for definition of performance events in Java. In this way JPMF decouples the occurrence of an event (cf. scope in IDM) from the measurement of performance data (cf. probes in IDM). Primarily designed for Java, JPMF is programming-language-specific.

4. CONCLUSION

In this paper, we presented a novel instrumentation description model (IDM) for the purpose of measurement-based performance evaluation of managed-language-based applications. IDM allows to describe instrumentation and monitoring instructions in an application-independent and monitoring-tool-independent way, enabling portability of instrumentation instructions and, thus, a more generic applicability of different measurement-based performance evaluation approaches.

Though the presented IDM is the current state of ongoing research, it gives an insight on a common, generic instrumentation description language, constituting a potential standard for different, performance-oriented instrumentation tools like Kieker [9] or SPASS-meter [4]. So far, we used IDM to create generic instrumentation descriptions for measurement-based, automated diagnostics of performance problems [10]. To this end, we created a Java instrumentation engine [1] which directly uses IDM descriptions as input.

Furthermore, we are currently working on adapters for Kieker which translate an IDM instance to corresponding Java or .NET instrumentation configurations for Kieker. We plan to extend the current version of IDM by further scopes and probes to provide a more comprehensive language.

5. REFERENCES

[1] Aim: Adaptable instrumentation and monitoring. visited: October 2014. http://sopeco.github.io/AIM.

[2] T. Aotani and H. Masuhara. Scope: an aspectj compiler for supporting user-defined analysis-based pointcuts. In *AOSD'07*, pages 161–172. ACM, 2007.

[3] S. Chiba and K. Nakagawa. Josh: an open aspectj-like language. In *AOSD'04*, pages 102–111. ACM, 2004.

[4] H. Eichelberger and K. Schmid. Flexible resource monitoring of java programs. *JSS*, 93:163–186, 2014.

[5] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. G. Griswold. An overview of aspectj. In *ECOOP'01*. Springer, 2001.

[6] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J.-M. Loingtier, and J. Irwin. *Aspect-oriented programming*. Springer, 1997.

[7] L. Marek, A. Villazón, Y. Zheng, D. Ansaloni, W. Binder, and Z. Qi. Disl: a domain-specific language for bytecode instrumentation. In *AOSD'12*. ACM, 2012.

[8] Q-ImPrESS. Java performance measurement framework, January 2011. http://www.q-impress.eu/wordpress/wp-content/uploads/2011/01/D6.1-Annex-Guidelines-and-Tool-Manuals_Final_version.pdf.

[9] A. van Hoorn, J. Waller, and W. Hasselbring. Kieker: A framework for application performance monitoring and dynamic software analysis. In *ICPE'12*. ACM, 2012.

[10] A. Wert, J. Happe, and L. Happe. Supporting swift reaction: automatically uncovering performance problems by systematic experiments. In *ICSE'13*. IEEE, 2013.

[11] M. Woodside, G. Franks, and D. C. Petriu. The future of software performance engineering. In *FOSE'07*. IEEE, 2007.