Model-based Performance Instrumentation of Distributed Applications

Jan Schaefer[1,2], Jeanne Stynes[2], Reinhold Kroeger[1]

[1] Wiesbaden University of Applied Sciences
Distributed Systems Lab
Kurt-Schumacher-Ring 18, D-65197 Wiesbaden, Germany
{jan.schaefer,kroeger}@informatik.fh-wiesbaden.de

[2] Cork Institute of Technology
Department of Computing
Rossa Avenue, Bishopstown, Cork, Ireland
jeanne.stynes@cit.ie

Abstract. Problems such as inconsistent or erroneous instrumentation often plague applications whose source code is manually instrumented during the implementation phase. Integrating performance instrumentation capabilities into the *Model Driven Software Development* (MDSD) process would greatly assist software engineers who do not have detailed knowledge of source code instrumentation technologies. This paper presents an approach that offers instrumentation support to software designers and developers. A collection of instrumentation patterns is defined to represent typical instrumentation scenarios for distributed applications. A UML profile derived from these patterns is then used to annotate UML models. Based on suitable code generation templates, the annotated models are transformed into instrumented source code for different instrumentation APIs. A prototypical implementation, including an adaptation to Web services, was evaluated in a lab environment.

1 Introduction

In recent years, *Model Driven Software Development* (MDSD) has become increasingly popular[3] because several MDSD tools have reached a sufficient level of maturity. In MDSD, code generators are used to generate application source code from technical models based on transformation templates. Using this approach, source code for specific types of platforms and applications can be created efficiently. Today, several Open Source MDSD code generator frameworks are available and used in professional projects, in particular *AndroMDA*[4] and *openArchitectureWare*[5] have become popular in recent years. Because of diverse application requirements, extensions containing specific templates and UML profiles for these frameworks are constantly being developed. So far, these extensions

[3] http://www.voelter.de/data/articles/cgn.pdf
[4] http://www.andromda.org
[5] http://www.openarchitectureware.org

cover mainly middleware infrastructure aspects (e.g. for EJB, CORBA, Spring, Hibernate). Extensions supporting mandatory application management aspects like security and performance are still rare.

Performance is an important aspect of applications, even more so in heterogeneous distributed systems. Thus, continuous performance tests, performance validation – especially after modifications or redesigns – and *Service Level Management* (SLM) at runtime are necessary tasks during the lifecycle of applications. This can be achieved by applying *Performance Instrumentation*, which can be defined as the process of adding non-functional code to an application to provide performance analysis information at runtime.

Instrumentation is usually performed during the implementation and testing phases, when software developers analyse the application's source code. Once relevant positions have been identified (e.g. based on their importance for the application), instrumentation statements are inserted into the source code. Manual instrumentation always carries the potential of errors or unwanted side-effects: the instrumentation might be incomplete, too detailed (and therefore slow down the application) or too sparse. Tools supporting developers during the instrumentation process greatly reduce the probability that these common mistakes will occur. More importantly, enhanced tool support removes the developers' need to acquire detailed knowledge of the applied instrumentation technology before the instrumentation process is carried out.

An instrumentation has to be integrated with the underlying application architecture, which can become a time-consuming and difficult task if the performance aspect is not considered until the end of the development process. Unfortunately this occurs frequently even though application performance and responsiveness are major acceptance factors for end users. Once applications are deployed or have evolved over time, there may exist immobile technical or architectural dependencies that must be observed if monitoring capabilities or performance-related changes have to be implemented. Such dependencies can be avoided by implementing performance monitoring capabilities as early as possible, preferably before the first pieces of source code are written. In this paper, logging and performance measurements (e.g. execution times of work units) are considered as the primary goals of performance instrumentation.

Based on Pooley's definition [1] of *Software Performance Engineering*, a performance engineering process integrating instrumentation with the MDSD methodology can be defined as follows: UML application models are annotated with an UML instrumentation profile. The resulting annotated models are transformed into instrumented source code using specific templates developed for a MDSD framework. This enables software designers to define performance monitoring capabilities in UML application models during the design phase without detailed knowledge of the instrumentation technologies that are used in the generated code. If standardised instrumentation APIs such as *log4j*[6] and *Application Response Measurement*[7] (ARM) are targeted during code generation,

[6] `http://logging.apache.org/log4j`
[7] `http://www.opengroup.org/arm`

the runtime performance data produced by the instrumented application can be processed easily by enterprise management systems such as *IBM Tivoli*[8] or *HP Business Technology Optimization Software*[9]. Therefore, the focus of this paper is on modelling and transformation of performance annotations.

This introduction is followed by a presentation of the state of the art in application performance instrumentation in section 2. Section 3 introduces the unique instrumentation approach developed for this paper. A performance engineering process incorporating this approach is presented in section 4, followed by its prototypical implementation in section 5 and a case study in section 6. This paper closes with a conclusion and a look at possible future work in section 7.

2 Related Work

Instrumentation can be required and performed during almost any phase of an application's lifecycle. This section introduces common approaches to software-based instrumentation that support developers in the process.

Apart from the risk of erroneous instrumentation, manual source code instrumentation can lead to a possibly unwanted mixture of functional (business logic) and non-functional (instrumentation) source code. Thus, instrumentation approaches using *Aspect-Oriented Programming* (AOP), where no instrumentation code has to be written repeatedly once templates are created, have been implemented in recent years [2] [3]. However, aspect compilers such as *AspectJ* which are used by these approaches often support granularity at method invocation level only. Another drawback is the lack of correlation functionality (i.e., the absence of facilities for semantically related instrumentation points to reference each other). This is not a problem for independent logging instrumentation but, especially in distributed systems, end-to-end monitoring based on related measurements can be mandatory to track requests on their way through complex workflows. Also, current AOP-based instrumentation approaches can be used only from the implementation phase on.

Binary code instrumentation is necessary if the source code of the to be instrumented application is not available or must not be modified. This approach is often used in conjunction with the Java programming language [4], because Java offers standardised interfaces for modifications to bytecode even at runtime (e.g. engaging bytecode running in the *Java Virtual Machine* (JVM) [5] [6]). Although arbitrary positions in binary code can be addressed in general, this instrumentation approach suffers from similar limitations as AOP. Correlation facilities are not provided, and obviously this approach can only be used if binary code for the targeted application already exists. The abstraction ability of binary code (or machine code for that matter) is too limited because it is supposed to be a concrete (platform-specific) implementation of the application.

In recent years, the need for instrumentation led to the development of middleware frameworks that already contain fixed instrumentation capabilities as

[8] `http://www.ibm.com/software/tivoli`
[9] `http://www.managementsoftware.hp.com`

developed by the vendor. For example, IBM instrumented[10] their *DB2 Universal Database*[11] (version 8.2 or later) and *WebSphere Application Server*[12] (version 5.1.1.1 or later). And starting with Java 5, even the standard edition JVM contains *Java Management Extensions*[13] (JMX), which support state monitoring of applications at runtime.

Another approach suited to instrumenting framework-based client/server applications uses the widely supported *message handler* framework (also known as *interceptor* or *listener* framework). It is part of the CORBA [7] and *Java API for XML Web Services* (JAX-WS) [8] specifications and supported by application servers such as the *Apache Tomcat*[14] and *JBoss*[15] application servers. This approach relies on instrumented components that are plugged into the frameworks by configuration transparently to the application [9] [10]. This approach can even be combined with legacy middleware technologies if supported by a connector such as an *Enterprise Service Bus* (ESB).

All these instrumentation alternatives cannot be integrated with the MDSD process because they do not feature modelling capabilities. However, several approaches for integrating performance aspects with UML models have been developed. The definition of the *UML Profile for Schedulability, Performance, and Time*[16] (UML-SPT, now known as MARTE) sparked a vast collection of research projects with the intention to implement the SPE requirements [11]. So far, work based on the UML-SPT primarily focussed on systems with strict timing and performance constraints (e.g. real-time systems). The process of creating a complete application model with performance annotations for each component can become very time-consuming. Nevertheless, this is common practice, especially for developing embedded systems. This design detail may be mandatory for simulating and validating system properties prior to implementation [12], but it removes the advantage of relieving developers during the instrumentation process if the supporting solution increases the modelling effort drastically. In addition, the modelling effort required for complying with the UML-SPT is greater than what is required for basic application monitoring.

Performance prediction is based on models like *Petri Nets* [13] [14], *Queueing Models* or *Markov Chains* [15] [16]. These approaches focus on stochastic methods for predicting qualitative (correctness) and quantitative (performance) applications properties or even complex systems. Pooley proposes generating such models from UML models, which is used seldom in a traditional software development process as software designers and developers are usually unfamiliar with this task. Furthermore, in order to be of practical relevance, queueing models have to be calibrated based on runtime measurement data, which must be

[10] `http://www.ibmsystemsmag.com/i5/june05/features/9060p3.aspx`
[11] `http://www.ibm.com/db2`
[12] `http://www.ibm.com/websphere`
[13] `http://java.sun.com/javase/technologies/core/mntr-mgmt/javamanagement`
[14] `http://tomcat.apache.org`
[15] `http://www.jboss.org/products/jbossas`
[16] `http://www.omg.org/technology/documents/formal/schedulability.htm`

collected using some sort of monitoring anyway [17]. If the sample data used for this purpose is too limited or generally inappropriate, the results of the subsequent analysis will not reflect the real system behaviour. Thus, queueing models cannot replace but can complement a concrete instrumentation.

3 Model-based Performance Instrumentation

As mentioned in section 1, this paper focuses on the integration of logging and performance measurement capabilities with the MDSD process. This section presents the approach to integrate an abstract representation of instrumentation information with application models.

3.1 Instrumentation Patterns

Figure 1 presents a service invocation as an instrumentation scenario example which could be instrumented by defining two related execution time measurements. The server-side measurement corresponds to the execution time of the service ($t_{2\rightarrow3}$), the client-side measurement represents the response time visible to the client ($t_{1\rightarrow4}$). If these measurements were linked, they would allow an analysis of execution and response time of each processed request.

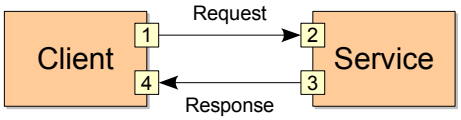

Fig. 1. Instrumentation Scenario

This paper defines a collection of *Instrumentation Patterns* representing abstract instrumentation scenarios. By referring to this pattern collection, software designers can determine possible instrumentation scenarios in their application models. The pattern collection offers additional guidance to designers because the relationships between determined instrumentation points might not always be recognisable.

The pattern collection can be split into two groups: *Basic Patterns* and *Complex Patterns* [18]. Basic patterns are the building blocks of complex patterns. In addition to the patterns introduced in this paper, new patterns can be defined based on either basic or complex patterns.

Instrumentation points and their purpose are described in more detail by their *Role* in a pattern. Each pattern defines a set of roles detailing the responsibilities of the associated points (e.g. instrumentation points can take either "start" or "stop" role in a measurement). An instrumentation point can be part of multiple basic patterns, which themselves can be part of multiple complex patterns.

The instrumentation of an application – the collection of all its pattern instances – can be seen as a directed graph: instrumentation points are vertices of this graph and their connecting edges can be annotated to further describe the relationships between the instrumentation points. A pattern (a subgraph) is described by one (basic pattern) or more (complex pattern) tuples. Basic instrumentation patterns describe simple workflow elements that can occur in applications. Figure 2 displays the three basic patterns defined by this paper.

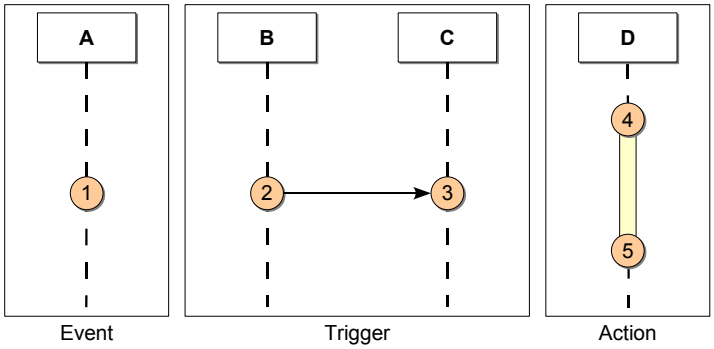

Fig. 2. Basic Instrumentation Patterns

The *Event Pattern* is the simplest instrumentation pattern. It is used for defining single, unrelated instrumentation points. Thus, it is usually represented by a log or status message (see example A in figure 2). The name of the only role in this pattern is *source*, because the point in this pattern is the source of the event.

The *Trigger Pattern* defines an event that sets off an arbitrary number of other events. A triggered event is causally dependent on its trigger event. The trigger event and the triggered event can be processed by a single system component or by multiple (distributed) components. `Triggers` support synchronous and asynchronous application execution scenarios. (see example B/C in figure 2). The are two roles in this pattern: *activator* and *receiver*. A trigger event also can be blocked (e.g. in a queue).

The *Action Pattern* defines two related `Events` (a start and a stop `Event`) which are processed by a single system component or a pair of related components. An `Action` spans a certain period of an application's execution time (see example D in figure 2) and can be seen as a specialisation of the `Trigger` pattern. The `Action` pattern also has two roles, namely *start* and *stop*.

The *RPC Pattern* shown in figure 3 is an example of a complex pattern. [19] contains a more detailed description of the pattern collection including the *Multitrigger* and *Sequence* patterns. The RPC pattern represents a synchronous or asynchronous message exchange. The activities on client and server component can be seen as related work units. Thus, the server-side activity is semantically

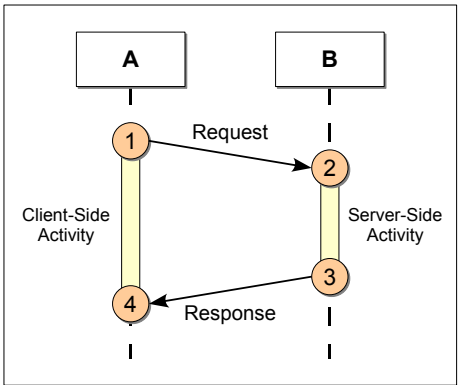

Fig. 3. RPC Pattern

nested in the client-side activity. This describes a classic *Remote Procedure Call* (RPC) interaction in which server activities are triggered by client invocations. This pattern is a composition of four basic patterns: two `Actions` (*client-side activity* and *server-side activity*) and two `Triggers` (*request* and *response*).

3.2 UML Instrumentation Profile

The abstract graphical pattern representation must be mapped to appropriate UML entities to enable software designers to use these patterns in application models. The *UML Instrumentation Profile* shown in figure 4 represents a mapping of the patterns introduced in section 3.1 to UML. The stereotypes in this profile contain abstract instrumentation information required for logging and performance measurements.

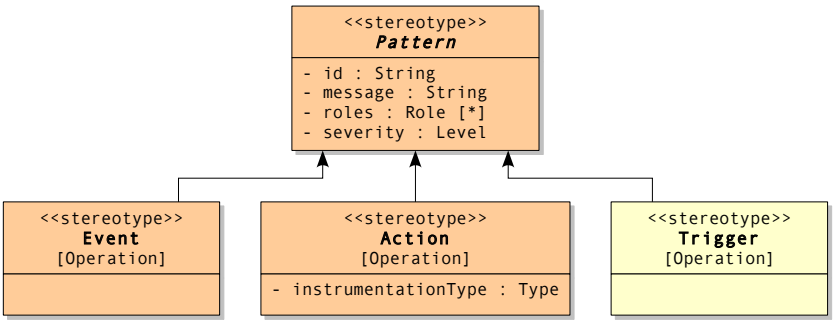

Fig. 4. Basic instrumentation stereotypes

The abstract *Pattern* stereotype contains shared tagged values, which are required by all basic and complex instrumentation patterns. The `id` attribute (or

tagged value) contains a unique (human-readable) name of the *Instrumentation Entity*, which can be either a point or a pattern. The `id` can be used to name the entity. The `message` attribute contains a message describing the entity. Depending on the instrumentation APIs provided by the code generator, the `message` might reemerge in the generated instrumentation code. `roles` is an enumeration literal containing the role(s) that an entity takes in patterns (see section 3.1). The role merely can be used to supply additional detail to instrumented entities which allows the code generator to generate specific instrumentation code (e.g. for a selected middleware platform). The `severity` is also an enumeration literal and defines the importance level of the output of this instrumented entity. Based on the common features of today's logging frameworks, the abstract pattern supports the levels `Info`, `Debug`, `Warn`, `Error`, `Fatal` and `Trace`. The default importance level is set to `Debug`.

The *Event* stereotype does not introduce additional tagged values. It can be attached to UML operations. During code generation, the `Event` is typically implemented by a logging statement.

The *Action* stereotype introduces an additional tagged value named `instrumentationType`. The instrumentation type is an enumeration literal that can be set to either `Logging` or `Measurement`. Depending on its value, source code for logging or response time measuring will be generated.

The *Trigger* stereotype and the complex stereotypes are not graphically representable in UML class diagrams, which have been investigated for this paper, using UML notation. This diagram type supports static associations between classes, but it is impossible to mark source and target instance of an operation invocation nor patterns spanning multiple UML entities. The case study in section 6 discusses this limitation further.

4 Performance Engineering Process

This section introduces a performance engineering process, illustrated in figure 5, that is compatible with the Software Performance Engineering approach described by Pooley in [1]. It is based on UML, the UML instrumentation profile as introduced in section 3.2 and the *openArchitectureWare* (oAW) MDSD code generation framework for Java source code generation, but the methodology can be transferred to other frameworks and programming languages easily. For illustration purposes, the subsequent description uses oAW-specific terminology.

Before the instrumentation process is started, the instrumentation profile is imported into the software designer's UML modelling tool. During creation and analysis of the UML application models, the designer can annotate designated instrumentation points using the stereotypes of the instrumentation profile. Once this process has been finished, the instrumented model is exported to XMI.

The code generation workflow of the oAW framework is configured to import and parse the instrumented model using the profile metamodel. The oAW code generator component generates pure Java source code for uninstrumented UML elements (using generic *JavaBasic Templates* provided by the *Fornax Plat-*

form[17]) and instrumentation code for instrumented elements (using custom *Instrumentation Templates* developed for processing stereotyped UML elements). Figure 5 also exemplifies a UML class extended with the *Event* stereotype and the resulting instrumented Java source code.

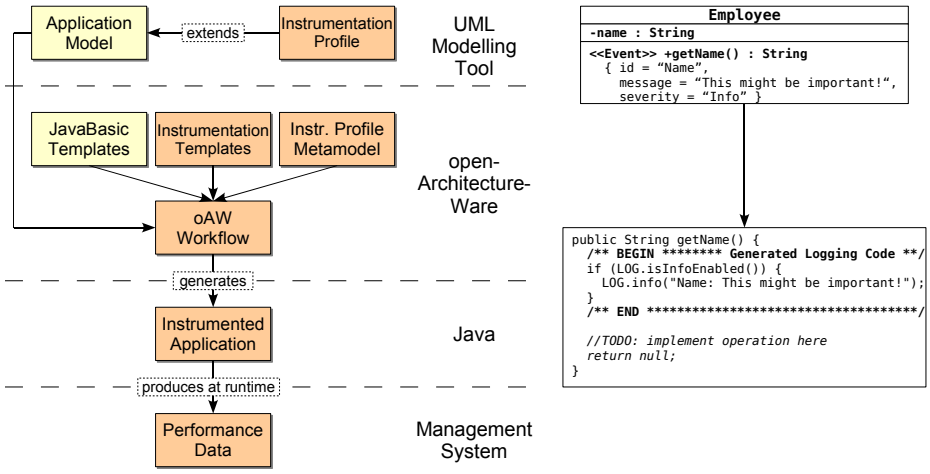

Fig. 5. Performance Engineering Process

Once the generated instrumented application source code is completed by implementing the application's business logic, the *Instrumented Application* generates *Performance Data* at runtime, which can be processed (i.e. displayed, analysed) using a *Management System*. In case common instrumentation APIs and libraries are used for generating the instrumentation code, existing enterprise management systems can be used for the analysis.

The approach presented in this paper does not dictate following the MDSD approach during development of all application components. Modelling and instrumentation can also take place in the beginning only, followed by more traditional source code-based development afterwards. However, the presented approach can be integrated seamlessly into a MDSD process.

5 Prototypical Implementation

An overview of the oAW-based code generation process has already been given in section 4. Custom instrumentation templates have been developed for generating instrumentation code. oAW features an AOP mechanism supporting the extension of existing templates. The instrumentation templates extend the Fornax JavaBasic templates just as the instrumentation profile extends UML. The oAW workflow presented in figure 5 is clarified by figure 6.

[17] `http://www.fornax-platform.org`

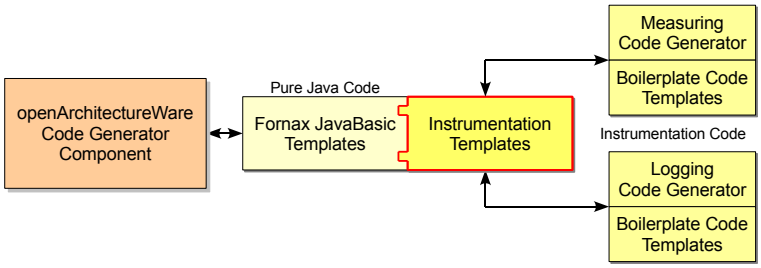

Fig. 6. oAW Code Generator Instrumentation Extension

The instrumentation extension developed for oAW contains templates that support code generation for the logging and time measurement instrumentation goals. The prototypical implementation creates a source code representation of the instrumentation patterns introduced in section 3.1 to the well-known log4j (for logging) and ARM 4.0 (for measuring) APIs. ARM is a widely acknowledged *Open Group* standard for performance measurements within distributed applications. Within ARM, *Response Times* are execution time measurements of work units termed *ARM Transactions* within distributed applications. To avoid dependencies on global time, each measurement has to start and end within the same process. However, the standard allows the correlation of semantically related measurements, even across host boundaries. For this purpose, ARM defines *ARM Correlators*, which are unique tokens assigned to each ARM transaction. ARM is capable of recording single ARM transactions, which is a requirement for the instrumentation of critical applications, and supports direct integration of applications with enterprise management systems. This creates a comprehensive end-to-end monitoring capability, including the measurement of application performance, availability, usage and end-to-end transaction response times. To effect this integration, ARM calls must be present in the application source code, which are processed by an ARM library during application execution. ARM defines C and Java APIs.

Both code generators retrieve the required instrumentation statements from textual code templates. These templates have been developed based on the *Velocity*[18] template engine, so the targeted instrumentation APIs can be exchanged without modifying the code generators' source code. For `Events`, logging statements are placed at the beginning of generated methods stubs; for `Actions`, two measurement statements are placed at beginning and end of method stubs. ARM measurement data map to the corresponding instrumented source code locations containing `start()` and `stop()` calls on the transaction object. Therefore, a response time value as defined in the ARM 4.0 API can only express the time span referenced by two instrumentation points located in the same application instance and thus on the same host.

[18] `http://velocity.apache.org`

Depending on the instrumentation stereotypes and tagged values of each instrumentation point detected in a parsed UML model, the instrumentation templates invoke the appropriate code generator for generating either measuring or logging statements. The positions, in which these statements are placed, are shown in figure 7.

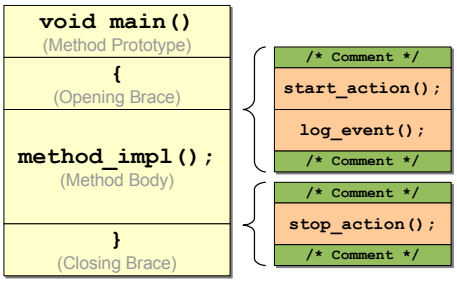

Fig. 7. Instrumentation Code Insertion Positions

6 Case Study: Web Services

The performance engineering process has been applied to several examples in a lab environment (i.e. without real-world application and work load). One example is presented in this section to demonstrate the applicability of the process to modern middleware-oriented applications and the flexibility of the developed prototype.

As discussed in section 3.2, complex patterns like the `RPC` pattern cannot be applied to UML class diagrams graphically. On the other hand, the `RPC` pattern as presented in section 3.1 is essential when instrumenting distributed applications. In order to solve this conflict, an adaptation has been developed which uses roles to textually represent the `RPC` pattern in UML class diagrams, so that the code generator can generate appropriate instrumentation code.

For the adaptation to Web service facilities, the client-side and the server-side `Actions` were outfitted with their respective roles (as introduced in section 3.1) in the UML diagram, which were then interpreted by the code generation templates appropriately. First, JAX-WS-based Web service communication, which is supported by major Web service frameworks such as *Apache Axis 2*[19], *Apache CXF*[20] and even *Java 6*[21], was analysed for facilities supporting ARM correlation of distributed measurements. Figure 8 shows the resulting exchange of a

[19] `http://ws.apache.org/axis2`
[20] `http://incubator.apache.org/cxf`
[21] `http://java.sun.com/javase`

correlation token (CT) between client and Web service based on the *Web service context* and *message handler* facilities. The generated instrumentation code inserts an ARM correlator into the Web service context, which is attached as metadata to the outgoing request by a message handler. On the service side, another message handler extracts the correlator and puts it into the context. The generated instrumentation code for the service then uses the received correlator as parent correlator for its ARM measurement.

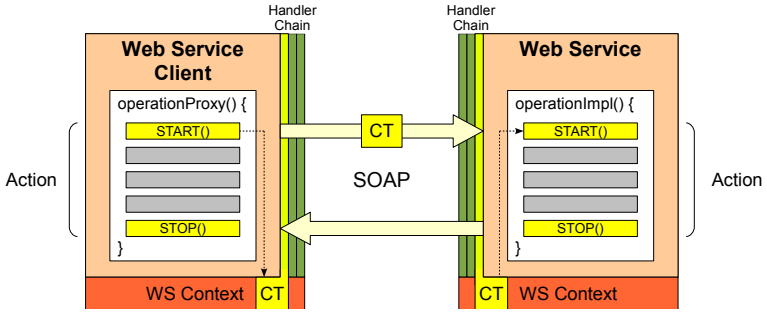

Fig. 8. JAX-WS Facilities in Web Service Interaction

The case study showed that the model-based instrumentation approach can be applied to middleware-oriented applications, although adaptation is required for each additional framework to be supported. The amount of modifications required for adapting the code generator, however, was small so this does not pose a grave disadvantage. A look at the MDSD template collection hosted by Fornax confirms that the adaptation requirement is a general limitation of MDSD code generation: generic templates result in generated source code that requires comprehensive manual additions (thus limiting the benefit of code generation), and specific templates are limited in their applicability. There simply is no generic yet flexible solution appropriate for a broad palette of instrumentation scenarios.

Although the prototype is based on and integrated with openArchitectureWare, the presented approach can also be implemented in alternative MDSD code generation frameworks, programming languages and middlewares. Depending on the extension capabilities of the target platforms, the oAW-independent Java code generators and velocity templates might even be reusable.

7 Conclusion and Future Work

This paper presented an approach to model-based performance instrumentation of distributed applications in accordance with Pooley and modern MDSD-based software engineering processes. The performance aspect has been integrated into the MDSD process so that software designers can continue to use their existing

UML modelling tools to instrument application models. With the collection of instrumentation patterns presented here in mind, designers are able to identify interactions within the models that are possible candidates for instrumentation. A drawback of the UML instrumentation profile based on these patterns is that it is a custom development. However, existing profiles for integrating performance annotations with UML models (such as the UML-SPT/MARTE) lacked essential features due to their emphasis on real-time systems modelling. The custom-designed profile defined here suffers from the risk of being outdated by standards developed in the future.

The prototypical implementation of the architecture was evaluated in a case study which demonstrated the overall usability and adaptability of the approach. The tests showed that comprehensive code generation can be achieved for specific usage scenarios (here: JAX-WS-based Web services) with only minor modifications to the otherwise generic templates. This can have a great impact on the productivity of a software project: a developer familiar with the environment executes the adaptations required for integrating a new communications framework, and all peers can use and profit from the generated instrumentation. Further evaluation of the methodology and the prototype is part of an ongoing research project which allows applying the approach presented here to an enterprise application.

Although template-based code generation only offers limited flexibility, projects such as the Fornax Platform, which concentrates on developing and providing extensions to widely used MDSD frameworks, help create a toolbox for MDSD which should contain something useful for almost any software development project. So far, the available extensions are mostly middleware-specific. Increasing acceptance and usage of MDSD technologies in professional software development, however, might spark the interest in extensions for generating source code for non-functional application properties like management and security, which could be combined with existing templates. This would effectively add an additional layer on top of the currently available communication- and infrastructure-centric templates.

As UML class diagrams are the most popular diagram type today, they were initially investigated for applicability of the instrumentation profile. The result showed that class diagrams are not ideally suited for instrumenting distributed applications. For example, dynamic interactions between distributed entities (e.g. Remote Procedure Calls) cannot be described sufficiently. But for developing the performance engineering process, class diagrams were the best choice, based on the fact that most available resources for MDSD frameworks rely on this diagram type. The evaluation of additional diagram types for integration with the instrumentation patterns and the profile (e.g. UML sequence and state diagrams) has already been started.

References

1. Pooley, R.: Software engineering and performance: a roadmap. In: ICSE '00: Proceedings of the Conference on The Future of Software Engineering, New York,

NY, USA, ACM Press (2000) 189–199
2. Krishnamurthy, R.: Performance Analysis of J2EE Applications Using AOP Techniques. (2004) http://www.onjava.com/pub/a/onjava/2004/05/12/aop.html.
3. Weimer, C.: IDE-gestützte Generierung von Quellcode zur Instrumentierung von Anwendungen. FH Wiesbaden (2005)
4. WO 03/062986 A1: Flexible and extensible java bytecode instrumentation system. Patent (July 2003)
5. Buytaert, D., Maebe, J., Eeckhout, L., Bosschere, K.D.: Building Java program analysis tools using Javana. In: OOPSLA '06: Companion to the 21st ACM SIGPLAN conference on Object-oriented programming systems, languages, and applications, New York, NY, USA, ACM Press (2006) 653–654
6. US 2002/0152455 A1: Dynamic instrumentation of an executable program. Patent (October 2002)
7. M. Wegdam, A.v.H.: Experiences with CORBA interceptors. (2000) http://www.comp.lancs.ac.uk/computing/rm2000/papers/20-aacentcweg.pdf.
8. Pulavarthi, R.: Writing a Handler in JAX-WS. (2006) http://java.sun.com/mailers/techtips/enterprise/2006/TechTips_June06.html.
9. Schmid, M., Thoss, M., Termin, T., Kroeger, R.: A Generic Application-Oriented Performance Instrumentation for Multi-Tier Environments. In: 10th IFIP/IEEE International Symposium on Integrated Network Management (IM2007), IEEE (May 2007) 304–313
10. Debusmann, M., Schmid, M., Kroeger, R.: Measuring End-to-End Performance of CORBA Applications using a generic instrumentation Approach. In Corradi, A., Daneshmand, M., eds.: Proceedings of the Seventh IEEE Symposium on Computers and Communications ISCC 2002, IEEE (2002)
11. Smith, C.U., Williams, L.G.: Performance and Scalability of Distributed Software Architectures: An SPE Approach (2002)
12. Gomez-Martinez, E., Merseguer, J.: A Software Performance Engineering Tool based on the UML-SPT. In: QEST '05: International Conference on the Quantitative Evaluation of Systems (Proceedings), IEEE Computer Society (2005) 247
13. Anglano, C.: Performance modeling of heterogeneous distributed applications. In: MASCOTS '96: Proceedings of the 4th International Workshop on Modeling, Analysis, and Simulation of Computer and Telecommunications Systems, Washington, DC, USA, IEEE Computer Society (1996) 64
14. J. Dehnert, J.F., Zimmermann, A.: Workflow Modeling and Performance Evaluation with Colored Stochastic Petri Nets (2000)
15. Bolch, G., Greiner, S., de Meer, H., Trivedi, K.S.: Queueing Networks and Markov Chains: Modeling and Performance Evaluation with Computer Science Applications, 2nd Edition. Wiley-Interscience (2006)
16. Theelen, B., Voeten, J., van Bokhoven, L., van der Putten, P., Niemegeers, A., Jong, G.: Performance Modeling in the Large: A Case Study (2001)
17. Xu, J., Oufimtsev, A., Woodside, M., Murphy, L.: Performance modeling and prediction of enterprise JavaBeans with layered queuing network templates. SIGSOFT Softw. Eng. Notes **31**(2) (2006) 5
18. Kroeger, R., Machens, H.: Trace Framework - Tracing in heterogenen Umgebungen. Technical report, Wiesbaden University of Applied Sciences (Nov 2002)
19. Schaefer, J.: Model-based Instrumentation of Distributed Applications. Master's thesis, Cork Institute of Technology (2008)