Universität Karlsruhe (TH)
Research University · founded 1825

Automated Extraction of Palladio Component Models from Running Enterprise Java Applications

Diploma Thesis at the
Institute for Program Structures und Data Organization
Chair for Software Design and Quality
Prof. Dr. Ralf H. Reussner
Fakultät für Informatik
Universität Karlsruhe (TH)

Author:
Fabian Brosig
Fabian.Brosig@stud.uni-karlsruhe.de

Advisors:
Prof. Dr. Ralf H. Reussner
Dr.-Ing. Samuel Kounev
Dipl.-Inform. Klaus Krogmann

Date of Registration: 2008-11-01
Date of Submission: 2009-05-20

I declare that I have developed and written the enclosed Diploma Thesis completely by myself, and have not used sources or means without declaration in the text.

Karlsruhe, 2009-05-20

Abstract To ensure that a software system meets its performance requirements during system operation, the ability to predict its performance under different configurations and workloads is highly valuable. For performance prediction, we need performance models. However, building predictive performance models manually requires a lot of time and effort. Current performance analysis tools used in industry mostly focus on profiling and monitoring transaction response times and resource consumption. Such tools often provide large amounts of low level data which is irrelevant to the models while important information about, e.g, the resource demands of individual components is missing.

In this thesis we develop a method for semi-automated extraction of Palladio Component Model (PCM) instances of Java EE applications based on monitoring data collected during operation. PCM is a domain-specific modelling language to describe performance-relevant information of a component-based architecture. We implement the developed method in a tool prototype. To obtain monitoring data we use state-of-the-art, industrial monitoring tools available for the current version of the Oracle WebLogic Server Platform. We evaluated the approach in the context of a case study with a real-world enterprise application (a beta version of the successor of the SPECjAppServer2004 benchmark). Even though the extraction of PCM instances require some intervention, the prototype we implemented provides a proof-of-concept showing how the existing gap between low level monitoring data and high level performance models can be closed.

Zusammenfassung Um die Einhaltung von Leistungsanforderungen während der System-Laufzeit sicherzustellen, ist die Fähigkeit, die Leistung eines Software-Systems unter verschiedenen Konfigurationen und Auslastungsgraden vorherzusagen, sehr wichtig. Für die Leistungsvorhersage werden Modelle (Leistungsmodelle) benötigt, die relevante Leistungseigenschaften des betrachteten Systems abbilden. Manuelles Erstellen solcher Modelle ist sehr zeit- und arbeitsaufwändig. Die meisten derzeitig in der Industrie verwendeten Werkzeuge zur Leistungsanalyse zielen auf das Messen von Ressourcenverbräuchen sowie Ausführungszeiten einzelner System-Funktionen ab. Viele Werkzeuge ermöglichen zwar die Erfassung von großen Mengen maschinennaher Daten, diese Daten sind jedoch nicht hilfreich bei der Erstellung der Leistungsmodelle. Wichtige Informationen über zum Beispiel den Ressourcenbedarf einzelner Komponenten werden nicht geliefert.

In dieser Diplomarbeit entwickeln wir eine Methode für eine semi-automatische, auf Messdaten basierende Extraktion von Palladio Component Model (PCM) Instanzen. Das PCM ist eine Modellierungssprache, die insbesondere auf Leistungsvorhersagen abzielt. Die entwickelte Methode implementieren wir als Prototyp. Die Messdaten erheben wir während der Laufzeit. Dafür nutzen wir aktuelle, in der Industrie verwendete Werkzeuge, die für die jetzige Version der Oracle WebLogic Server Plattform verfügbar sind. Der Ansatz wurde anhand einer Fallstudie mit einer realistischen Geschäftsanwendung evaluiert (mit einer Vorversion des Nachfolgers des SPECjAppServer2004 Benchmarks). Obgleich die Extraktion von PCM Instanzen Benutzereingriffe erfordert, liefert der Prototyp einen Machbarkeitsnachweis, wie die Lücke zwischen maschinennahen Messdaten und abstrahierenden Leistungsmodellen geschlossen werden kann.

Contents

List of Figures

List of Tables

1. Introduction

Performance is a critical factor for successful software projects [?]. Although hardware speed is continuously increasing, software performance problems are common since software system complexity and size are growing at a fast pace [?]. A widespread misconception is that performance problems can be addressed by simply throwing enough hardware at the system [?].

To avoid performance problems, it is important to analyze the expected performance characteristics of systems during all phases of their life cycle. The methods used to do this are part of the discipline called Software Performance Engineering (SPE). SPE is described as a "systematic, quantitative approach to the cost-effective development of software systems to meet performance requirements" [?]. At each stage of the software development process, SPE helps to estimate the level of performance a system can achieve and provides recommendations to realize the optimal performance level [?].

In this diploma thesis, the term performance is understood as the degree to which a software system meets its objectives for timeliness and the efficiency with which it achieves this [?]. Timeliness is measured in terms of meeting response time or throughput requirements and scalability goals. Thus, performance involves both timing behavior and resource efficiency. Request throughput and response times are not the only properties of interest. As part of SPE, resource consumption has to be considered as well. Especially with reference to green computing [?], minimizing the application's resource demands while keeping the application's time behavior at the required level is of increasing importance.

1.1 Motivation

To ensure that a software system meets its performance requirements, the ability to predict its performance under different configurations and workloads is highly valuable throughout the system life cycle. During the design phase, performance prediction helps to evaluate different design alternatives. At deployment time, it facilitates system sizing and capacity planning. During operation, predicting the effect

of changes in the workload or in the system configuration is beneficial. The alternative to performance prediction is to deploy the system in an environment reflecting the configuration of interest and conduct experiments measuring the system performance under the respective workload. Such experiments, however, are normally very expensive and time-consuming and therefore often considered not to be economically viable. Furthermore, experiments obviously require an implementation of the system or at least a running prototype to be available. To enable performance prediction we need an abstraction of the real system that incorporates performance-relevant data, i.e., a performance model. Based on such a model, performance analysis can be carried out.

Building predictive performance models manually requires a lot of time and effort [**?**]. The model has to represent performance-relevant parts of the system by reflecting the abstract system structure. In addition, model parameters like resource demands or system configuration parameters have to be determined.

Hence, an automated performance model extraction from a running system under consideration would be useful. During the development phase this would ease the evaluation of a prototype's performance behavior. During operation, an automatically extracted performance model can be applied for run-time performance management. If one observes an increased user workload and assumes a constant workload growth rate, performance predictions help answering when the system would reach its saturation point. For instance, for specific workloads one can predict resource utilization and service response times. In this way, the system operator can react *before* application performance fails to meet objectives like, e.g., service level agreements.

Current performance analysis tools used in industry mostly focus on profiling and monitoring transaction response times and resource consumption. Such tools often provide large amounts of low level data (e.g., physical memory utilization) that is irrelevant to the models while important information about, e.g, the resource demands of individual components is missing. The tools do not extract a performance model that can be used for further analysis. They normally do not help until a bottleneck actually effects the performance behavior. Currently, the monitoring data has to be aggregated and analyzed manually when deriving performance models. This is even complicated due to the inability of most tools to be configured to collect data only in selected parts of the application with an appropriate level of granularity.

1.2 Aim of the Thesis

The aim of this thesis is to develop a method for automated extraction of performance models of enterprise systems based on monitoring data collected during operation. We use state-of-the art industrial monitoring tools and attempt to provide an end-to-end solution for the extraction. We focus on the Java Platform, Enterprise Edition (Java EE) infrastructure and examine Java EE applications deployed under different application server configurations.

As performance model, the Palladio Component Model (PCM) is chosen. The PCM is an established "meta-model allowing the specification of performance-relevant information of a component-based architecture" [**?**]. For the specification and analysis of PCM instances an integrated modelling environment is provided. The tool *PCM*

Bench [**?**] enables the creation of PCM model instances and allows for deriving performance metrics from the models using analytical techniques or simulation.

In this thesis the following specific goals are pursued:

- Develop a tool that generates a PCM instance from a running enterprise Java application by means of data obtained through available monitoring tools. The generation will be conducted semi-automatically, i.e., it will require some limited intervention. We aim at a semi-automatic approach since the currently available monitoring tools do not provide all data needed for PCM instance generation. The idea is to provide a proof-of-concept showing how the existing gap between low level monitoring data and high level performance models can be closed.
- Conduct a case study with a real-world enterprise application in order to evaluate the applicability of the approach.
- Identify the issues that need to be addressed in order to enable complete automation of the model generation process.

Requirements for the tool prototype are: i) the accuracy of the performance predictions should be in a range of $\approx 20\%$, ii) given that the considered application is assumed to run, the monitoring overhead has to be taken into account. The performance of the tool prototype itself is of secondary importance.

1.3 Outline

Starting with a short description of the foundations in Chapter 2, we then explain our approach to extract PCM instances (Chapter 3) and give an overview of related work in Chapter 4. A detailed description of the developed extraction method is provided in Chapter 5. Chapter 6 describes the implementation of the tool prototype and is followed by Chapter 7 that contains a case study to evaluate the concept's applicability to real-world enterprise systems.

2. Foundations

We provide an overview of technologies, products and tools used in the thesis. Starting with a short description of Java EE, Oracle WebLogic Server (WLS) and its monitoring tools are introduced. Additionally, we provide information on the Windows 2008 performance monitoring tool. For performance modelling we use PCM. We evaluate our results on the basis of a benchmark application.

2.1 Java Platform, Enterprise Edition (Java EE)

Java EE is a standard for distributed, multi-tier enterprise Java software. It consists of several Java Community Process specifications concerning technologies for an "enterprise-class server-side development and deployment platform" [**?**]. It aims at reducing the cost and complexity of developing portable and scalable Java applications providing multi-tier services [**?**]. The current version is Java EE 5 [**?**], previous versions are denoted as J2EE.

A significant part of Java EE is the Enterprise JavaBeans (EJB) Architecture. It is a server-side framework for component-based Java applications. In order to enable development with focus on the business logic, the Java EE platform provides an infrastructure that takes care of technical issues like transactions, security, persistence, clustering and much more [**?**]. The EJB 3.0 specification supports three types of beans: i) session beans that encapsulate business logic in the form of services executed through synchronous invocation, ii) message-driven beans (MDBs) that encapsulate business logic executed to process asynchronous messages sent through a message-oriented middleware, iii) entity beans that model persistent data. For session beans, there are two subtypes: stateless session beans and stateful session beans. Stateless session beans have no *conversational state*. This means that the conversational context between a client and a bean spans only a single method call. The consequence is that, from the client point of view, "all instances of the same stateless session bean class are equivalent" [**?**]. For a stateful session bean, the conversational context between a client and a bean may span multiple method calls. Thus, an instance of a stateful session bean has a state which can represent the history of the current conversation with the currently active client.

In EJB 3.0, from the bean developer's perspective, a session bean is a designated Java class implementing methods of a designated Java interface. Such an interface is denoted as *business interface*, the methods it contains are called *business methods*. The bundle of Java interface and implementation class is deployed in an *EJB container*. The EJB container is responsible for managing the EJBs, addressing the technical issues mentioned above.

In the following, we describe how the encapsulation is realized. Note that the given description is only a simplified one, for a detailed description, see [**?**]. When a client application calls a session bean's business method, the client never calls this method directly. Instead, the client always refers to the corresponding business interface. Internally in the EJB container, the client call invokes a method of a wrapper class. The wrapper class is generated by the container when the EJB is deployed. It implements the bean's business interface and wraps the actual bean class. This adapter pattern allows the container to perform middleware tasks, e.g., EJB lifecycle management or transaction management, when business methods are called.

Implementations of the Java EE specification are also denoted as Java EE application servers. The implementation we consider in this thesis is provided by the Oracle Corporation.

2.2 Oracle WebLogic Server (WLS)

The Oracle WebLogic Server (WLS) 10.3 is an application server compliant with the Java EE 5 specification [**?**]. It comes with the Oracle JRockit Java Virtual Machine (JVM). The WebLogic version 10.3 is the first version after the acquisition of BEA Systems by Oracle in 2008. A Gartner Research Note [**?**] judges BEA Systems to be one of the leaders in the market of enterprise application servers in the second quarter of the year 2008. The WebLogic product family would show a "strong presence in mission-critical business software" [**?**]. Additionally, the list of published results of the SPECjAppServer2004 benchmark shows a strong presence of WebLogic Server among Java EE server systems [**?**].

A WebLogic Server setup is organized in WLS domains. A WLS domain is a group of WebLogic Server resources. For example, a WLS domain may comprise eight WLS instances on eight machines but it may also comprise only two WLS instances on one machine. In our context it is sufficient to understand a WLS domain as a group of WLS instances.

2.2.1 WebLogic Diagnostics Framework (WLDF)

WLS 10.3 provides a monitoring and diagnostics framework called WebLogic Diagnostics Framework (WLDF) [**?**]. WLDF enables monitoring of WLS instances and application-specific resources providing diagnostic data.

The two main features we make use of are the *data harvester* and the *instrumentation engine*. The data harvester can be configured to collect detailed diagnostic information about a running WLS and applications deployed within its containers like, e.g., the amount of free Java heap memory or the number of pooled bean instances of a

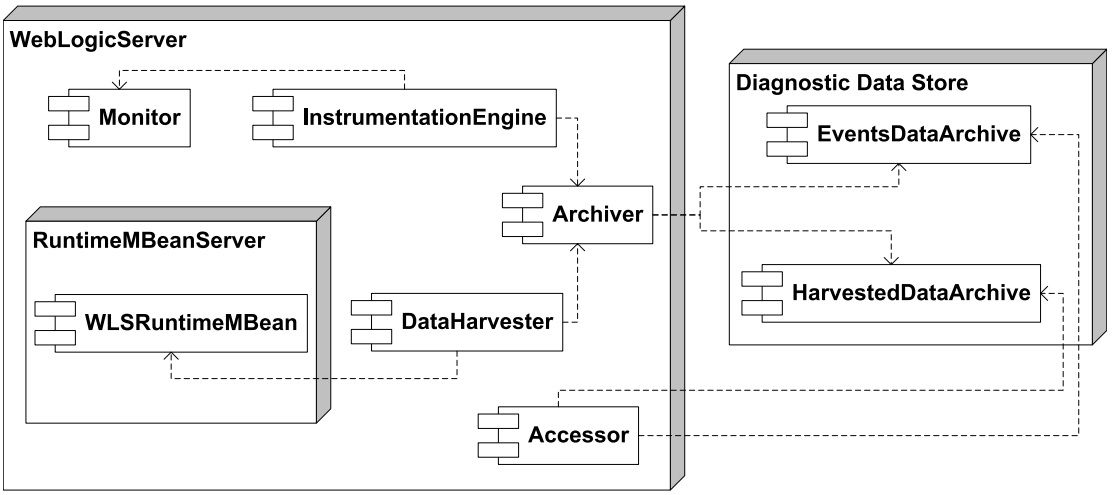


Figure 2.1: Simplified Overview of the WLDF Architecture [**?**]

specific stateless session bean. The instrumentation engine allows injecting diagnostic actions into server or application code at defined locations. In short, a location can be the beginning or end of a method, or before or after a method call. Depending on the diagnostic actions, event data creation is triggered each time the specific location is reached during processing. Both data harvester data as well as event data are persisted by an *archiver* that can be accessed later for further analysis.

2.2.1.1 WLDF Architecture

The WLDF architecture is depicted in Figure 2.1. There are two data generators for WLDF: the instrumentation engine and the data harvester. The instrumentation engine gathers information through *monitors* instrumenting server or application code. The data harvester gathers information by reading data from Runtime Managed Beans (MBeans) at a specified sampling rate. Runtime MBeans are objects that expose the state of the server's resources. These objects are registered at an MBean server. The MBeans can also be read via the Java Management Extensions (JMX) API. The latter "is a standard [...] for managing and monitoring applications and services" [**?**].

The diagnostic data is sent to the archiver that persists the data coming from the instrumentation engine in the *events data archive* and the data coming from the harvester in the *harvested data archive*. The diagnostic data store can be accessed by the *accessor* component.

2.2.1.2 Instrumentation Engine

The instrumentation engine weaves new code into existing Java byte code fragments to monitor run-time behavior. The injected code triggers a specific action when it is reached during execution. In WLDF, *monitors* are used to specify the actions to be performed and at which locations the actions should be injected. There are system-level monitors and application-level monitors, depending on whether the locations at which actions are injected reside in server code or in application code.

Monitors

WLDF distinguishes three types of monitors:

- Standard monitor: Both the actions and the locations are predefined and fixed.
- Delegating monitor: The locations are predefined and fixed, the actions are configurable. Examples: The predefined locations of monitor `EJB_After_SessionEjbBusinessMethods` are method exits of EJB business methods which are part of the application code. The predefined locations of the monitors `JDBC_Before_Statement_Internal` and `JDBC_After_Statement_Internal` are method entries and method exits in the WLS JDBC driver wrapper which is part of the server code.
- Custom monitor: Both locations and actions are configurable. It is not possible to define locations in server code, locations can only be defined in application code.

Locations

Locations, where WLDF diagnostic actions can be injected, are specified by a *pointcut* and a location type. In WLDF, a pointcut specifies a set of method call or method execution *joinpoints*. A method call joinpoint refers to a point where a specific method is called. A method execution joinpoint refers to a point where a specific method is implemented. Thus, a joinpoint is always attached to a method. A location type determines "the position relative to a joinpoint where the diagnostic action will take place" [**?**]. A joinpoint together with a location type defines a location.

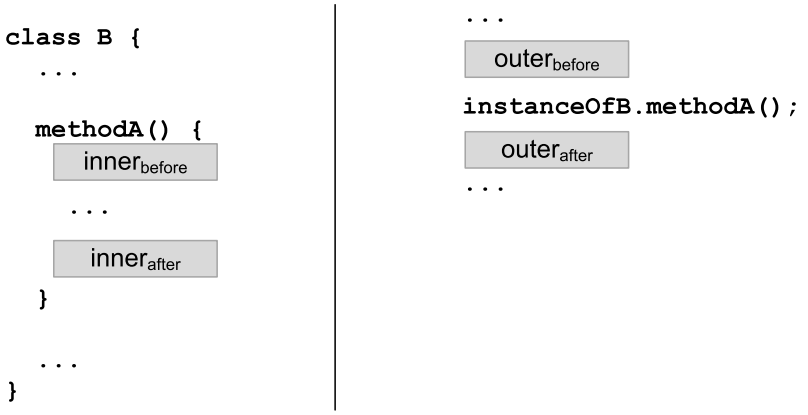
The concept of pointcuts and joinpoints stems from the *aspect-oriented programming* (AOP) paradigm. In fact, the WLDF instrumentation engine internally uses AOP. AspectJ [**?**] enables AOP for Java. In terms of AspectJ, a WLDF location type corresponds to an AspectJ *advice*.

Figure 2.2 illustrates how locations can be specified. With a pointcut specification the user selects a set of methods and specifies whether the points where the methods are invoked (`call`) or the method bodies (`execution`) are of interest. The location types `before`, `after`, and `around` determine the location to be before the joinpoints, after the joinpoints or both before and after the joinpoints. Example: The pointcut specification `execution(void B methodA())` together with location type `before` would specify the location $\text{inner}_{\text{before}}$ in method `B#methodA`.

When specifying pointcuts, the user can select the set of methods not only by full-qualified method name specifications but also by sophisticated qualifying expressions. For example, the following pointcut

- `execution(public * com.foo.bar.* initialize(...))`

matches method executions of all public `initialize` methods in all classes in package `com.foo.bar` and its subpackages. The `initialize` methods may return values of any type, including void, and may have any number of arguments of any types [**?**].



Pointcut type	Locations for location type		
	`before`	`after`	`around`
`execution`	$\text{inner}_{\text{before}}$	$\text{inner}_{\text{after}}$	$\text{inner}_{\text{before}}$, $\text{inner}_{\text{after}}$
`call`	$\text{outer}_{\text{before}}$	$\text{outer}_{\text{after}}$	$\text{outer}_{\text{before}}$, $\text{outer}_{\text{after}}$

Figure 2.2: Schema for specifying instrumentation locations in WLDF.

Actions

Opposed to pure aspect oriented approaches, within WLDF of WLS 10.3, actions that are subject to insertion are pre-defined. The following list presents four diagnostic actions that are most relevant in the context of this thesis.

- `TraceAction`: Used with location types `before` and `after`. "Generates a trace event [record] at the affected location in the program execution" [**?**].
- `DisplayArgumentsAction`: Like `TraceAction`, but captures also method arguments respectively method return values (depending on the monitor location type).
- `TraceElapsedTimeAction`: Used with location type `around`. Generates two event records: "one before and one after the location in the program execution. When executed, this action captures the timestamps before and after the execution of an associated joinpoint" [**?**] and computes the elapsed time with nanosecond granularity.
- `MethodInvocationStatisticsAction`: Used with location type `around`. When executed, this action computes the elapsed time of the associated joinpoint and aggregates the measurement with previous measurements of the same joinpoint. The aggregated data is used to calculate statistics of, e.g., count of invocations, average elapsed time, and standard deviation of elapsed time. The statistics are kept in memory, i.e., this action does not generate any event records.

Event Records

Besides the action `MethodInvocationStatisticsAction`, all available diagnostic actions generate event records. The event records are described by a single data

Field	Description
Record id	The numeric identifier that unambiguously identifies an event data record.
Timestamp	The timestamp when the event data record has been created, in milliseconds.
Context id	The diagnostic context identifier which uniquely identifies a request.
Transaction id	The transaction identifier (if available).
Action type	The diagnostic action.
Server name	The name of the server where the event record has been created.
Monitor name	The monitor that triggered the event record.
Class name	The name of the class where the action has been executed.
Method name	The name of the method where the action has been executed.
Method description	The parameter signature of the method where the action has been executed. The parameter signature comprises input parameter types as well as the return type in Java bytecode notation. Example: The parameter signature of method `void mymethod(int[] p1, boolean p2, String p3)` is `([IZLjava/lang/String;)V`

Table 2.1: WLDF event record fields [**?**].

structure. Table 2.1 shows the fields that are considered most relevant for the model extraction discussed in this thesis.

When instrumentation is enabled, WLDF attaches a *context id* to a request entering the system. This context id, which is unique in a WLS domain, travels along with the request, even as the request crosses thread boundaries and JVM boundaries. The context id lives for the whole life cycle of the request.

Additionally, the event record data structure contains fields that are only filled by a specific diagnostic action. Table 2.2 shows both how elapsed time and how method arguments are stored in an event record.

Deployment

System-level monitors can be added and removed dynamically during run-time. The adding and removing of monitors concerning the application-level requires a redeployment of the application, i.e., an interruption of the running application. This interruption can be avoided, if the *hot-swap* feature is enabled. However, for productive systems it is unusual that this feature is enabled. Note that WLDF monitors can be enabled and disabled. While in both cases diagnostic actions are injected, only those actions that belong to an enabled monitor are active. Enabling and disabling application-level monitors do not require a redeployment of the application.

Field	Description	
	`TraceElapsedTimeAction`	`DisplayArgumentsAction`
Payload	Elapsed time processing the associated joinpoint, in nanoseconds.	-
Arguments	-	Method input arguments, when the action is attached to a monitor with location type `before`. The method arguments are given as a comma-separated list of the actual parameters's String representations.
Return value	-	The method's return value, when the action is attached to a monitor with location type `after`. The return value is given as String.

Table 2.2: Action-specific event record fields [**?**].

2.2.1.3 Data Harvester

The data harvester gathers attribute values of MBean instances. The values are sampled, the information is read at a configurable sampling rate. The harvester can be configured to track the system state and performance because it supports harvesting of Runtime MBean instances. For instance, it can be configured to observe the amount of free Java heap memory, or the number of pooled bean instances of a specific stateless session bean.

Field	Description
Record id	The numeric identifier that unambiguously identifies a harvested data record.
Timestamp	The timestamp when the harvested data record has been created, in milliseconds.
Server name	The name of the server where the harvested data record has been created.
MBean type	The type of the harvested MBean as String.
MBean name	The name of the harvested MBean instance.
Attribute name	The name of the harvested attribute of the harvested MBean instance.
Attribute value	The value of the harvested attribute of the harvested MBean instance as String.

Table 2.3: WLDF harvested data record fields [**?**].

The data harvester is configured by specifying a sampling period (in seconds) and a set of MBean attributes to observe. Then, at each sampling point in time, for each

gathered attribute value, a harvested data record is created and sent to the archiver. Table 2.3 shows an excerpt of the fields of a harvested data record: Similar to an event data record, there are fields for the record id, the timestamp and the server name. The remaining fields describe the MBean instance where the attribute value is taken from.

2.2.1.4 Archiver and Accessor

Both the event data and the harvested data are persisted in a diagnostic data store. WLDF allows to configure a file-based store or to attach a database as storage location. Generally, file stores offer better throughput than a database store and generate no network traffic [**?**]. For diagnostic data of one single WLS instance, the archiver guarantees to store the diagnostic records in the correct order, i.e., chronological order with ascending record ids.

The data accessor offers access to the diagnostic data by an interface that abstracts from the concrete storage configuration. This interface provides a query language to select particular data. The query language supports filtering of diagnostic records on field values, but does not allow for customized ordering [**?**].

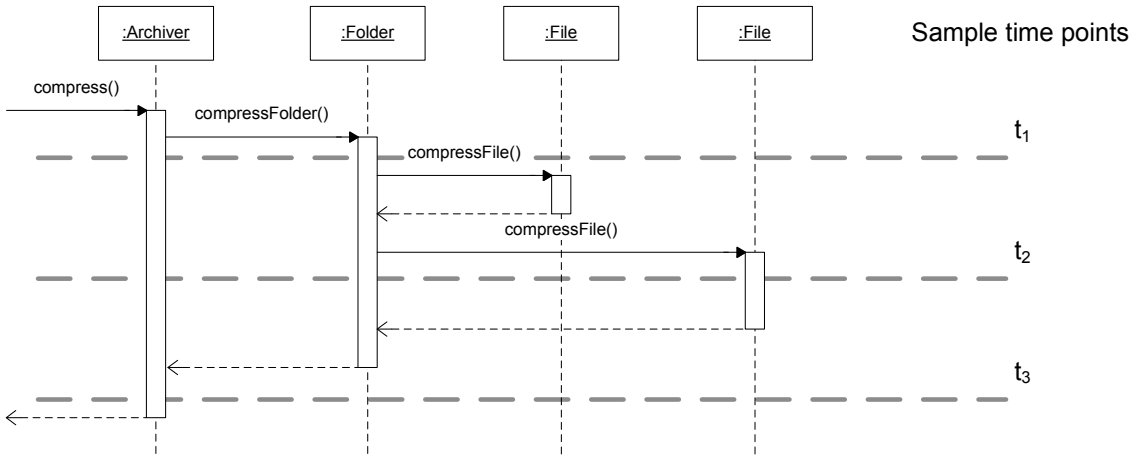
2.2.2 JRockit Mission Control

For monitoring at the JVM level, Oracle JRockit Mission Control comprises a set of tools running on the Oracle JRockit JVM [**?**]. The tool JRockit Runtime Analyzer (JRA) can be triggered on one or more Java processes. JRA produces recordings about the JVM, the application it is running and generates a report for offline analysis. Among other things, the recorded data includes information about method sampling (synonym for method profiling).

The JRA method sampling feature, that we apply to extract resource demands (see Section 5.6), is described in the following. The user can specify a sampling rate: the length of the time interval between the samples can be configured down to 1 millisecond. At one sampling point in time, JRA logs the methods currently executed by active threads that are running in the Java process JRA was invoked on. If the user configures JRA to consider stack traces, the tool logs the method stack according to a user-specified trace depth. The JRA recording contains a sample count for each logged method, and allows to visualize call traces depending on the configured stack trace depth.

The example in Figure 2.3 depicts how JRA considers method stacks during method sampling. The UML sequence diagram shows an execution path of the method `compress` of a simple archiver. The points $t_1, \ldots, t_3$ indicate when JRA collects method samples. The table describes which method samples, depending on the trace depth, are collected (methods surrounding `Archiver#compress` are omitted in this example). At t_2 for instance, with a trace depth of 3 the method `Archiver#compress` is sampled. With a trace depth of 2 the method `Archiver#compress` would not be sampled at t_2. This leads to the conclusion that with a sufficient trace depth, a method is always sampled at a sample time point when the considered method itself or one of its successors are executed. Obviously, this only holds if the execution takes place in a thread that is actually considered at the sample time point.

It is claimed that JRA method profiling has "typically less than 2 percent overhead" [**?**]. To reduce the overhead, at a sampling point in time JRA does not



Sample time points	Collected method samples	
	trace depth=2	trace depth=3
t_1	`Archiver#compress` `Folder#compressFolder`	`Archiver#compress` `Folder#compressFolder`

t_2	`Folder#compressFolder`
	`File#compressFile`
		`Archiver#compress`
		`Folder#compressFolder`
		`File#compressFile`

t_3	`Archiver#compress`	`Archiver#compress`

Figure 2.3: Example: Method sampling with different trace depths.

consider each active thread but selects the threads to monitor at random. However, we expect the overhead to increase if the user configures a high sampling rate and a high stack trace depth.

2.3 Windows Performance Monitor (perfmon)

The Windows Performance Monitor (perfmon) [**?**] is a monitoring tool that allows observation of resource utilization. We use the version 6.0.6001 shipped with Windows Server 2008. The feature we make use of is the observation of specific system resources during a specified time interval. The observations are performed via sampling. The minimal supported overall sampling rate is limited to one sample per second. We measure CPU utilization, disk utilization, network utilization and memory utilization.

The CPU utilization is represented by the fraction of time the processor is non-idle. If the CPU has multiple cores, the utilization is given as the average utilization of all cores. In perfmon, we use the performance counter `% Processor Time`. According to the description provided by the tool, the "calculation of whether the processor is idle is performed at an internal sampling interval of the system clock (10ms)". Note that this rate is independent of the overall sampling rate mentioned above.

For the disk, there are performance counters `% Disk Read Time`, `% Disk Write Time` and `% Disk Time`. However, these metrics are problematic because the percentage may exceed 100 percent [**?**] and therefore we only use them as indicator for disk load. A valid quantification of disk utilization is obtained with the perfmon

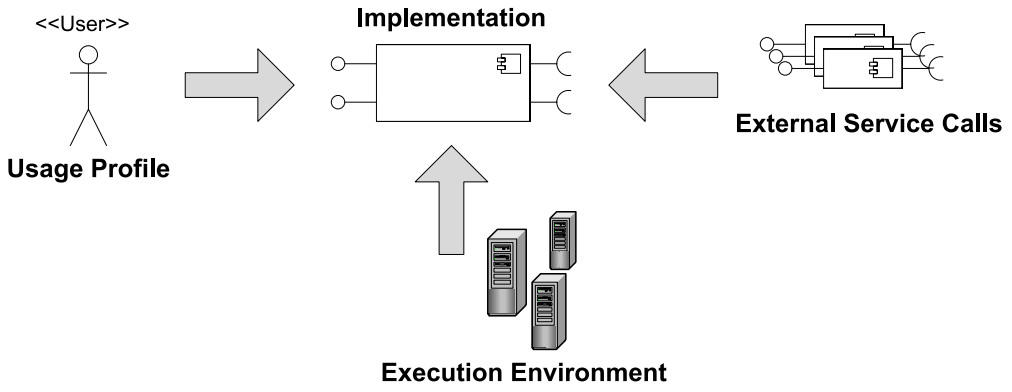


Figure 2.4: Performance-influencing factors of a component.
Performance-influencing factors of a component [**?**].

performance counter for the disk idle time [**?**]. With this measure, we quantify the busy time percentage by 100 minus the value of `% Idle Time`.

In order to monitor if the network or the memory are saturated we refer to the resource overview perfmon provides. There, the network as well as the memory utilization are provided. Note that we do not use these values for further computation, we only use them to check if the network or memory are bottlenecks.

2.4 Palladio Component Model (PCM)

The Palladio Component Model (PCM) is a modelling language for Quality-of-Service (QoS) predictions, in particular performance predictions. It is a "meta-model allowing the specification of performance-relevant information of a component-based architecture" [**?**] to build predictive performance models.

The PCM is aligned to the Component-Based Software Engineering (CBSE) process (see [**?**]) by providing a domain-specific modelling language for each developer role. Four developer roles, namely component developer, system architect, system deployer and business domain expert are distinguished [**?**].

The components are specified and implemented by the component developer. A Palladio component specification mainly comprises a specification of required and provided interfaces as well as performance-relevant implementation details of provided services. System architects assemble components to build architecture models of applications, whereas system deployers create a resource model representing the system environment and allocate components of the architecture model to resources. Business domain experts finally develop PCM usage models describing the expected system usage by means of different usage scenarios.

Enabling performance predictions requires the consideration of performance influencing factors. As illustrated in Figure 2.4, in order to capture the time behavior and resource consumption of components, four factors have to be taken into account. Obviously, the component's implementation affects the performance. Additionally, the component may depend on external services whose performance has to be considered as well. Furthermore, the deployment context has an impact on the component's

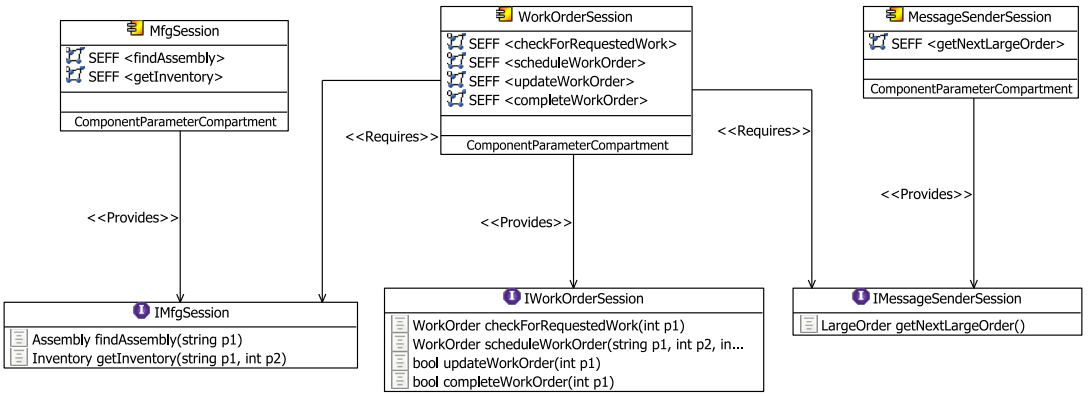


Figure 2.5: Example: PCM repository.

performance: Both the way how the component is used and the execution environment have to be taken into account for the prediction. The PCM allows for these factors by supporting parameter dependent specifications [**?**, **?**, **?**]. Thus, different hardware resources, assembly contexts and usage contexts can be considered when conducting performance analysis.

The *PCM Bench* tool [**?**] facilitates the development and analysis of PCM instances. Once a PCM instance is built, different performance solvers can be triggered. For example, PCM offers an analytical queueing network solver as well as a pure simulation-based approach named *SimuCom*. In this thesis we use the SimuCom since "it supports all features of the PCM models" [**?**] whereas other solvers do currently not cover the entire set of PCM constructs.

Nevertheless, if the PCM instance is not developed during the software development process, obtaining a PCM instance afterwards is a complex task. Besides manual modelling [**?**], automatically extracting PCM instances from existing software systems remains a challenge (see Chapter 4). In the remainder of this section, we present the sub-models representing the domain-specific modelling languages for the different developer roles.

2.4.1 Repository

The repository contains the components, the interfaces the components require or provide, and data types required to specify the services the interfaces comprise. Relations between components are not expressed directly, they are only connected through interfaces respectively through roles that refer to interfaces.

The repository shown in Figure 2.5 consists of three components whereas each component provides one interface. Component `WorkOrderSession` requires the interfaces `IMfgSession` and `IMessagerSenderSession`, the other two components do not require any interface.

To abstractly describe the internal behavior of a service that is provided by a component, PCM introduces the concept of a service effect specification (SEFF). For the purpose of performance modelling, there is a specialization of a SEFF, denoted as resource demanding SEFF (RDSEFF). A RDSEFF abstractly depicts the component service's control and data flow by mapping the relationship between the

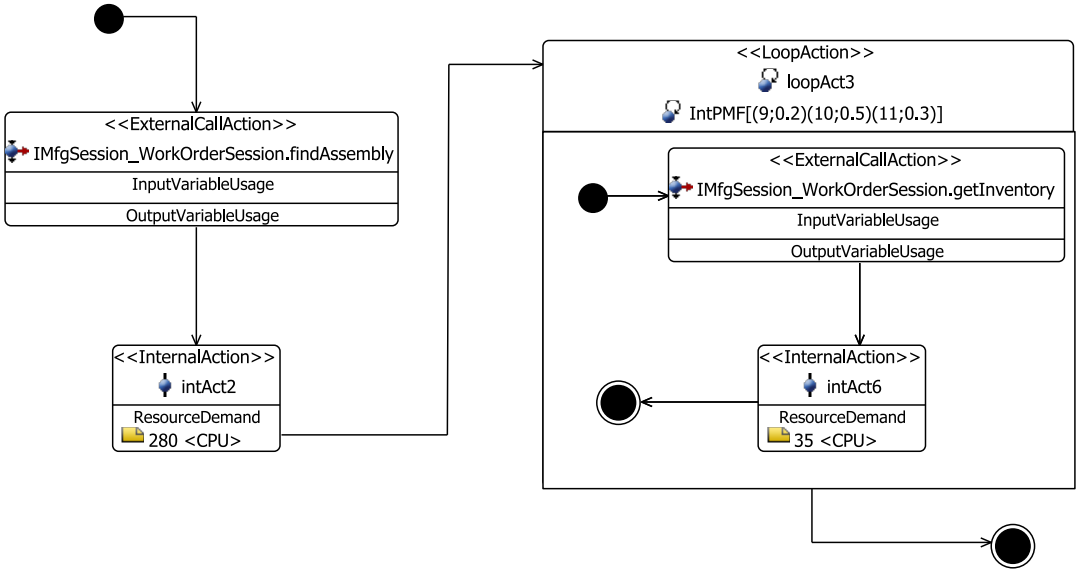


Figure 2.6: Example: RDSEFF of component `WorkOrderSession` (see Figure 2.5).

provided service and the required services. The notation of a RDSEFF resembles the notation of UML activity diagrams: There are activities, in PCM notation called actions, that can be assembled to characterize the service effect. The capability i) to enrich actions by annotations specifying resource demands and ii) to specify transition probabilities and resource demands depending on the service's formal parameter is a distinctive feature of a RDSEFF.

The RDSEFF in Figure 2.6 describes the service `IWorkOrderSession# scheduleWorkOrder` as implemented in component `WorkOrderSession`. Starting with an external call of service `IMfgSession#findAssembly` and an internal action requiring CPU resources, there is a loop whose loop iteration number is specified as probability function. The loop body contains an external call to `IMfgSession#getInventory` and a further internal action that is enriched by a CPU resource demand annotation. For simplicity, this example does not make use of parameter-dependent constructs. As shown in the repository in Figure 2.5, service `IMfgSession#findAssembly` demands a parameter of type string as input, but the presented RDSEFF model does not specify this parameter, i.e., the input parameter is not considered to influence the performance of the described service implementation.

2.4.2 System

The system architect uses components from the repository and assembles them to build a system model. In fact, the architect creates so-called assembly contexts that reference components. Without the indirection via assembly contexts it would, e.g., not be possible to model components that are deployed on multiple resources. In PCM, a system is understood as a composed structure like a component that consists of other components. The PCM system model therefore contains not only connectors between assembly contexts, but also connectors between assembly contexts and the system boundary in order to determine the interfaces the system provides respectively requires.

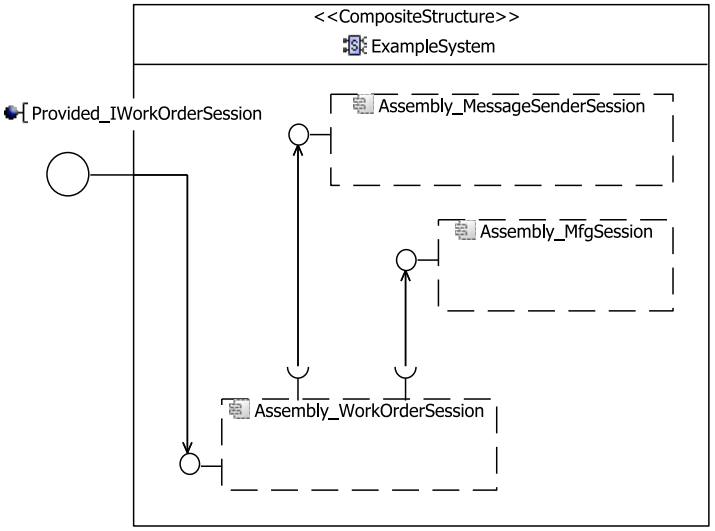


Figure 2.7: Example: PCM System (see also Figure 2.5).

Figure 2.7 illustrates the system model `ExampleSystem` that provides the interface `IWorkOrderSession`. Internally, the system consists of three assembly contexts that are connected according to their specification in the repository in Figure 2.5. In our context, it is sufficient to understand an assembly context as a component instance. Nevertheless, note that this is a simplification.

2.4.3 Resource Environment and Allocation

In PCM, the execution environment of a component instance is represented by a resource environment on the one hand and a mapping of component instances to resources by means of an allocation model on the other hand. The resource environment consisting of resource containers has to be specified at an abstract level. Palladio distinguishes two types of resources. There is a passive resource type with semaphore semantics (e.g., connection pools) and a processing resource type that is characterized by a processing rate (e.g., CPUs). For the connection of two resource containers, PCM offers a linking resource type represented as specialization of the processing resource type.

Note that in the current PCM version passive resource types are not modelled in the PCM resource environment model but in the components.

2.5 Benchmark SPECjAppServer2004_Next

The benchmark we consider for evaluation is a beta version of the successor of SPECjAppServer2004 [1]. We denote that version as SPECjAppServer2004_Next. Note that it is a synonym specific to this thesis.

[1]SPECjAppServer is a trademark of the Standard Performance Evaluation Corp. (SPEC). The SPECjAppServer2004 results or findings in this publication have not been reviewed or accepted by SPEC, therefore no comparison nor performance inference can be made against any published SPEC result. The official web site for SPECjAppServer2004 is located at http://www.spec.org/osg/jAppServer2004.

SPECjAppServer2004_Next is a Java EE benchmark developed by SPEC's Java subcommittee for measuring the scalability and end-to-end performance of Java EE-based application servers. The benchmark's workload is generated by an application that is modelled after an automobile manufacturer. The application represents a real-world e-business system comprising customer relationship management (CRM), manufacturing and supply chain management (SCM) as business scenarios.

Figure 2.8 shows the architecture of the benchmark as it is described in the benchmark documentation. The actual benchmark application consists of an orders domain, a manufacturing domain and a supplier domain. The domains are deployed on the considered Java EE application server, they are implemented with EJBs according to the EJB 3.0 specification. The domains interact with a database server via Java Database Connectivity (JDBC), or more precisely, via the persistence framework Java Persistence API (JPA). The point-to-point communication between the domains is implemented using the Java Message Service (JMS) queues, i.e., it is asynchronous. The workload for the orders domain is triggered by dealerships, the workload for the manufacturing domain is triggered by manufacturing sites. In the context of the benchmark, both dealerships and manufacturing sites are simulated by the benchmark driver. The supplier domain in turn is triggered by the manufacturing domain. Connected suppliers are simulated by the supplier emulator. Whereas suppliers and the supplier domain communicate only via Web services, the orders domain is accessed by Java Servlets. The manufacturing domain is accessed either thorugh Web services or EJB calls, i.e., Remote Method Invocation (RMI). As illustrated, the system under test spans both the Java application server and the database server. The emulator and the benchmark driver are required to run outside the system under test to not affect benchmarking results.

The benchmark driver knows five benchmark operations. A dealer may browse through the catalog of cars, purchase cars or manage his dealership inventory, i.e., sell cars or cancel orders. A manufacturer may create vehicles (via Web service or RMI).

Internally, the following business logic is implemented: Whenever a dealer creates an order that is considered a large order, the large order is sent to the manufacturing domain. A large order is an order with more than 20 vehicles. The manufacturing domain processes large orders by scheduling work orders. New work orders can also be scheduled without outstanding large orders. To process a work order, components are required. If the manufacturing site's inventory does not contain enough components, purchase requests for new components are sent to the supplier domain. There, purchase orders are sent to selected suppliers. The suppliers in turn inform the supplier domain when the components are delivered. This delivery information is then routed to the manufacturing domain. When a work order is completed, the orders domain is informed about the fulfilled order. Note that, in fact, work order processing is implemented to be independent from the availability of components. A work order can also be completed without any notification from the supplier domain. For the benchmark, it is important to create some work, it is not required to implement an accurate business process.

We use the SPECjAppServer2004_Next application to evaluate our approach to performance modelling in the context of a case study (see Section 7.4). We consider the benchmark representative since it is based on a real-world, complex and state-of-

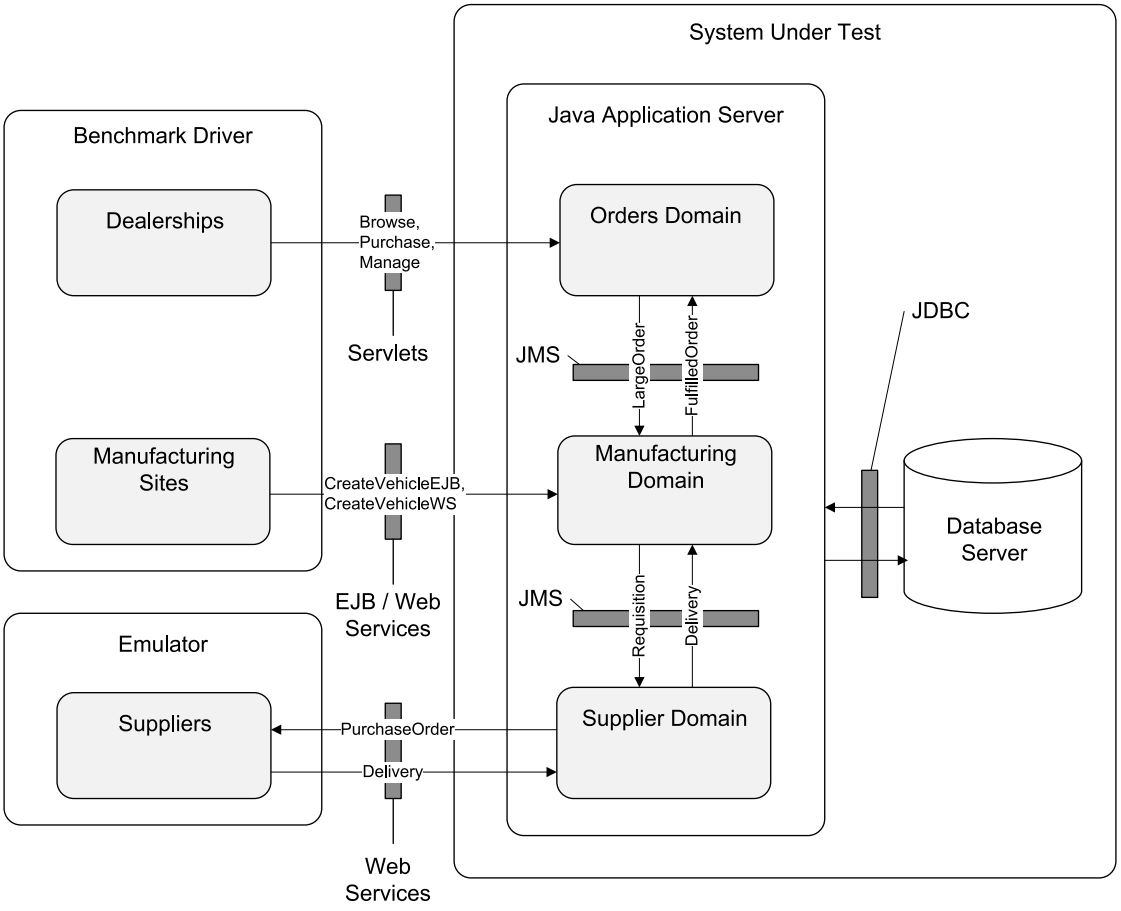


Figure 2.8: SPECjAppServer2004_Next architecture.

the-art system. Previous versions of that benchmark have already been successfully applied for research purposes [**?**, **?**, **?**, **?**]. The fact that SPECjAppServer2004_Next is the fourth version in the line of SPEC benchmarks targeting at Java EE application servers shows that this benchmark line is well adopted by the industry.

3. Approach

In order to enable performance prediction for Java EE applications, we extract PCM instances during run-time. This chapter briefly describes how we intend to achieve this goal, a detailed description of our method is given in Chapter 5. In the following we provide an overview of how we extract the application's architecture and resource demands (Sections 3.1 and 3.2).

3.1 Extract the Application's Architecture

Given a software application, extracting its architecture requires identifying its building blocks and the connections between them.

3.1.1 Componentization

In our context, componentization is the process of breaking down the considered software application into components. Thus, componentization is a part of software architecture reconstruction. Reverse engineering of software architectures is a large research field [**?**] which is outside the focus of this thesis. This thesis does not pursue automatic detection of components and component boundaries, it is assumed that information about the component boundaries is available. Such information can be obtained in different ways, e.g., specified manually by the system architect or extracted automatically through static code analysis (see Section 4.1.2).

The component definition underlying the PCM is: "A software component is a unit of composition with contractually specified interfaces and explicit context dependencies only. A software component can be deployed independently and is subject to composition by third parties" [**?**].

Considering the notion of components in Java EE it is evident that the relationship between components in terms of the software architecture and EJBs is not necessarily required to have a one-to-one cardinality [**?**]. We consider EJBs as components or elements of components. From an architectural point of view, one basic component can also consist of more than one EJB (without being modelled as composite component). Hence, when using the term "component", the component boundaries have to be explicitly.

3.1.2 Extract Control Flow

The control flow between the identified components and the control flow inside the components is extracted by call path tracing. Therefore, we analyze monitoring data consisting of event records obtained through instrumentation. In order to trace single requests, the event records have to be grouped and ordered. The set of groups represents the set of equivalence classes according to the following equivalence relation. Let a, b be event records obtained through instrumentation. Then a relates to b ($a \sim b$) if and only if a and b were triggered by the same system request. This is well-defined because an event record is triggered by exactly one system request. In the following, equivalence classes are denoted as *call path event record sets*. Ordering the elements of a call path event record set in chronological order results in a *call path event record list*. From this list a call path can be obtained.

For event record grouping, the WLDF instrumentation engine provides the concept of a diagnostic context id. For event record ordering, the WLDF instrumentation engine provides an event record id and event record timestamps (see Table 2.1).

Dynamic control flow analysis by call path tracing requires a representative workload during instrumentation. Only paths that are exercised can be captured. If the workload is not representative, e.g., does not cover all relevant components and service calls of the application under consideration the resulting model will be incomplete. This is a disadvantage compared to an analysis approach that is based on static information, e.g., on the application's code base. If there is a representative workload, the proposed analysis approach has the advantage to expose the "effective architecture" [**?**].

3.2 Extract the Application's Resource Demands

We estimate the application's resource demands from measurements we conduct during run-time. Measured values are, e.g., sampling counts in a fixed time interval, utilization of resources or response times. The measurement overhead has to be considered, since measuring itself may affect the system performance. This is often referred to as *Heisenberg Effect*.

Determining the resource demands of a transaction involves identification of demanded resources and quantification of resource-specific demands. In the example in Figure 3.1, we assume transaction tx_1 to use two processing resources P_1 and P_2 (e.g., CPU and I/O).

Figure 3.2 illustrates the processing resources demanded by tx_1. First resource P_1, then resource P_2 is used. For both processing resources, there are waiting queues. When a resource is busy (for example due to previous concurrent requests), incoming requests are put in the waiting queue. The more a resource is utilized, the longer the average waiting time in the corresponding queue is.

The timeline in Figure 3.3 depicts an exemplary execution of transaction tx_1. The example shows that the transaction's response time $t_4 - t_0$ consists of the waiting times for P_1 and P_2 and the resource demands at P_1 and P_2 we are interested in. In order to extract a transaction's resource demand for a given resource, we follow two approaches: i) estimate the resource demand using the response time or ii) estimate the resource demand based on the measured resource utilization and transaction

```
transaction tx1 {

  // use resource P1
  ...
  // use resource P2
  ...

}
```

Figure 3.1: Example: Resource demanding transaction tx_1.

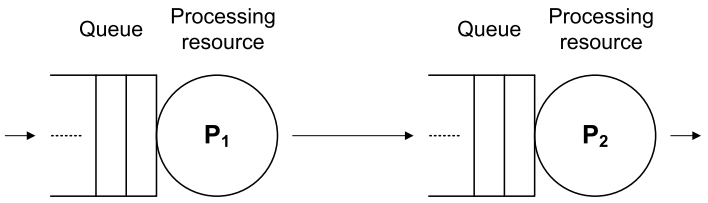


Figure 3.2: Example: Resources P_1, P_2 as demanded by transaction tx_1.

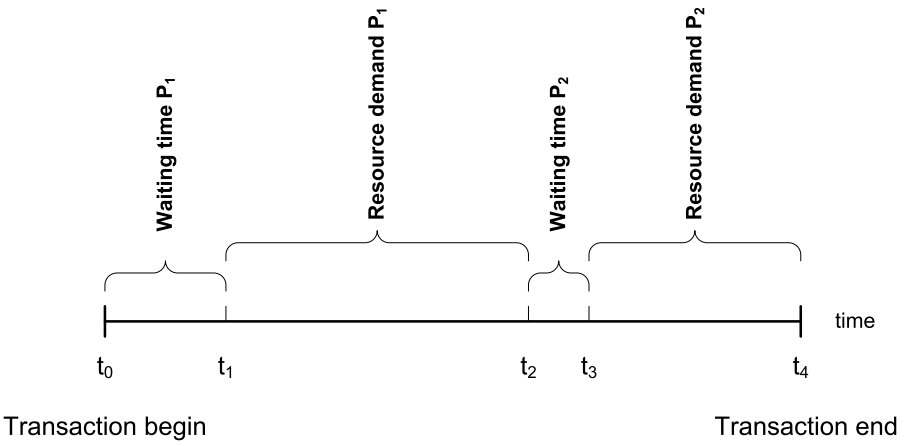


Figure 3.3: Example: Processing transaction tx_1.

throughput by means of the *Service Demand Law* [**?**]. The two approaches are explained in more detail below:

- **Response time.**
  - **Approach.** Estimate the resource demand using the response time.
  - **Assumptions.** The considered resource dominates the overall response time. The waiting time in the queue is insignificant compared to the actual processing time we are interested in.
  - **Applied on example from Figure 3.3.** Let the transaction of interest be tx_1 and the resource of interest be P_1. The response time $t_4 - t_0$ would include both waiting times and the usage of other resources. The time interval $t_2 - t_0$ would include the waiting time for P_1. The desired resource demand is $t_2 - t_1$, but normally this interval cannot be measured directly.
- **Service Demand Law.**
  - **Approach.** Compute the resource demand by the Service Demand Law. We introduce the notation:

T	Length of observation period.
$\#tx$	Number of completed transactions during T.
$\mathcal{U}$	Mean resource utilization during T as a fraction between 0.0 and 1.0.
$\mathcal{X}$	Throughput describing the number of completed transactions per time unit.
$\mathcal{D}$	The transaction's resource demand.

    If T, $\#tx$ and $\mathcal{U}$ are available, then the throughput $\mathcal{X}$ can be computed by $\mathcal{X} = \frac{\#tx}{T}$. Finally, the resource demand $\mathcal{D}$ is estimated using the Service Demand Law [**?**]. The law states that the average resource utilization $\mathcal{U}$ divided by the transaction's throughput $\mathcal{X}$ is the resource demand of the transaction:

    $$\mathcal{D} = \frac{\mathcal{U}}{\mathcal{X}}$$

    Note that this law refers to a single transaction and a single resource.
  - **Assumptions.** In the entire observation period, the entire utilization of the considered resource is produced by the considered transaction.
  - **Applied on example from Figure 3.3.** Let the transaction of interest be tx_1 and the resource of interest be P_1. If it could be guaranteed that only tx_1 is running during the observed time interval, this approach would be applicable.

Both approaches for resource demand extraction have strong assumptions. Since we observe the application under consideration during run-time, changing the execution environment to fulfill those prerequisites is not practicable. Hence, the challenge is to investigate how the approaches described above can be applied under weaker conditions.

- **Response time.** The challenge is to obtain the response time in a way, that the measured time intervals correspond to the actual resource demand. In order to minimize the error injected by resource waiting times, the resource's contention has to be low. In order to minimize the error that is injected by the usage of resources that are not under consideration, the measurement points have to be defined accordingly.

- **Service Demand Law.** Even if it is assumed that metrics like transaction throughput and resource utilization are easy to measure (with negligible errors), the application of the Service Demand Law is challenging. In order to ensure that the resource utilization only stems from the considered transaction, there are two options: i) It is assured that only the transaction of interest runs during the observation period. ii) The total resource utilization can be apportioned between the workload produced by the considered transaction and the workload produced by other transactions in a way that there is a fraction which maps to the workload as it is produced by the considered transaction. System monitors normally provide only total resource utilization statistics. We follow the approach of partitioning the resource utilization since we cannot assume to have the system's workload under control. We expect many transactions to run at the same time (a *transaction mix*). Hence, the total resource utilization has to be partitioned between the running transactions. However, in practice there is some resource usage (sometimes called *system overhead*), that cannot be assigned to only one specific transaction. "It is usually hard to find a way to distribute the unattributed resource usage" [**?**] among the transactions. Thus, the system overhead introduces an error when determining resource demands. Normally, the system overhead is almost constant, it does not scale with increased system load. That is why the resources should be "reasonably utilized" when measuring, e.g., 50%. The higher the resource utilization the lower is typically the relative error introduced by the system overhead.

  Menascé provides some methods for apportioning the total resource utilization in [**?**, **?**]. System workload is partitioned in so-called *workload classes* since real workloads are "viewed as a collection of heterogeneous components" [**?**]. There, the workload is partitioned among, e.g., different applications or workload classes that are derived based on resource consumption characteristics. We aim at a more fine-grained workload partitioning.

Since the measurements obviously depend on the execution environment, the extracted resource demands are platform-specific. That means, each time the execution environment changes, resource demands have to be extracted again. Nevertheless, platform-specific measurements result in a calibrated model that promises better predictions than a model with estimated platform-independent resource demands.

4. Related Work

For the purpose of performance prediction we extract performance models, more precisely PCM instances, from running enterprise Java applications using monitoring data. The monitoring data is obtained through WLDF. The following sections present related work concerning the automated extraction of performance models and general application monitoring (particularly run-time monitoring of Java applications).

4.1 Automated Extraction of Performance Models

In this section we show existing approaches for the automated extraction of performance models. As performance models, mainly LQN models and PCM instances are considered. Furher approaches to performance prediction are presented in subsection 4.1.3.

4.1.1 Layered Queueing Network (LQN) Models

Hrischuk et al. [**?**] and Israr et al. [**?**] both reverse engineer performance models using traces obtained via instrumentation. The approach is not restricted to Java systems, it is intended for general distributed systems that can simultaneously execute several distributed operations. In [**?**], *angio traces* are recorded. They are considered as special traces since each distributed operation gets assigned with a unique *angio dye id*. For message ordering, trace event timestamps are used. Once such traces are recorded, a graph representing the message flow is generated. Using a rule-based graph analysis approach the graph is then transformed into a LQN model whereas different interaction types such as synchronous and asynchronous messaging are detected and represented accordingly.

Since Israr et al. claim that such "traces are difficult to obtain in practice" [**?**], an approach based on a less restricted trace data format is proposed. It "uses conventional trace data which is available from many tracing tools" [**?**] and does not rely on angio traces containing an angio dye id that is propagated through the system.

Instead, so-called *eventInfo* properties of message traces are used to achieve message correlation. Here, eventInfo is not required to uniquely identify messages. However, it should provide information that, together with observed timestamps, makes a robust message correlation possible. Furthermore, Israr et al. generate LQN models considering different interaction types using an "algorithm which scales up linearly for very large traces" [**?**] in contrast to the algorithm presented in [**?**].

The injection of the diagnostic context id in WLDF resembles the concept of the angio dye id. Thus, our tracing approach corresponds to [**?**]. The target model of [**?**] and [**?**] is a LQN model and therefore not component-based in contrast to our performance model. Furthermore, we collect resource utilization information that goes beyond call path tracing whereas the LQN models extracted by Hrischuk and Israr still need to be enriched by resource demands.

4.1.2 PCM Instances

The reverse engineering approach presented in [**?**] is based on static analysis. The prototype called ArchiRec extracts information about EJBs by examining their code and their deployment descriptors [**?**]. The focus is placed on identifying the components the system is made of and using this information to automatically derive the system architecture. To evaluate component candidates, code coupling metrics are iteratively applied.

Inspired by [**?**], Java2PCM [**?**] aims at automatically retrieving RDSEFFs including parametric dependencies by means of source code analysis. The analysis "reconstructs an abstraction of the control and data flow" [**?**] of a service by identifying relevant actions. In order to obtain explicit parametric dependencies, a data flow analysis is applied. The dependencies are made explicit by "creating boolean expressions for branch conditions and arithmetic expressions for loop iteration numbers, which [...] reference [...] only parameters" [**?**].

Our approach uses run-time data for PCM instance generation. However, the above two approaches can be used to identify the component boundaries and obtain the information needed in order to automatically refactor the code according to the schema we propose in 5.4. Thus, these approaches can be considered as complementary to our approach.

In an ongoing project, Krogmann intends to reverse engineer PCM instances for Java applications using both static analysis and dynamic analysis. The component architecture is obtained via ArchiRec and RDSEFFs are obtained by applying machine learning algorithms on run-time monitoring data [**?**]. By monitoring service call frequencies and parameter values at the interface level, the black-box property of components is preserved. The monitoring data then serve as input for genetic programming that aims at recovering intra-method control flow and explicit parametric dependencies [**?**]. With regard to the resource demands, ByCounter [**?**] and microbenchmarks are used to abstract from concrete timing values [**?**]. ByCounter instruments the application bytecode for "dynamic counting of executed Java bytecode instructions" [**?**] and method invocations. Combined with benchmarking target execution platforms on bytecode instruction level whereas parameter influences are taken into account, performance predictions are then made possible. In [**?**], the approach is successfully applied in a case study. Particularly in view of the machine

learning algorithms's outputs and the determination of abstract resource demands further studies are of great interest.

While our approach uses WLS and WLDF, Krogmann's approach is execution platform independent. Both approaches assume a representative workload can be provided for the dynamic analysis. We extract the effective architecture and provide statistics about method arguments related to the control flow but do not obtain explicit parametric dependencies. Krogmann resolves explicit parametric dependencies using genetic programming. Since we collect timing values, we are able to obtain performance models that are calibrated for one execution environment and therefore the models can be expected to allow us predictions of higher accuracy. Furthermore, we profit from using WLDF as it allows us to obtain resource utilization information both at the system level and application level.

Another ongoing project concerning the generation of PCM instances is called Q-ImPrESS [**?**]. For the purpose of reverse engineering performance models, both static and dynamic analysis will be applied. "Static analysis is used to extract the static structure from the source code. The gathered information is then enhanced with measurements taken on existing and running systems using dynamic analysis." [**?**]. Furthermore, the usage profile will be derived from run-time monitoring data. However, Q-ImPrESS focuses on performance prediction at design time.

4.1.3 Other Measurement-Based Approaches to Performance Prediction

Denaro et al. [**?**] and Chen et al. [**?**] identify that the middleware functionality, such as transaction and persistence services, dominates distributed system performance. The middleware is considered as a key factor for performance problems in component-based applications.

For distributed software applications Denaro proposes performance testing instead of performance modelling. To allow performance prediction, application-specific performance test cases are developed to be executed on available middleware platforms. It is emphasized that the test case generation is also possible in the early stages of the software development process. Therefore, Denaro introduces an approach where performance test cases are derived from architecture designs.

While Denaro at al. propose the observation of application-specific test cases, Chen et al. propose an application-independent approach to benchmark middleware: A simple benchmark processing typical transaction operations is used to extract a performance profile of the underlying component-based middleware and to construct a generic performance model. This allows software architects to experimentally discover acceptable application configurations. However, application-specific behavior is not modelled explicitly.

Eskenazi et al. [**?**] describe another approach: A component operation's prediction model is described as a function over the operation's signature type to the resource demand whereas the signature type consists of performance relevant parameters. Via regression methods, the function is derived from resource demand measurements conducted while the specific component operation is executed in a testbed under performance-relevant use cases. The performance of a system request is then

computed by composing single operations's performance functions with regard to the control flow. Due to the required testbed, "factors affecting the perceived performance of a software component like influences by external services" [?] are neglected in this approach. In addition, to ensure the measurements's validity, the testbed must be stable during the development process.

Compared to our approach, the approach presented by Eskenazi et al. can be applied also at early architectural phases. However, while we extract the PCM instance during run-time of an application, that approach requires separate measurements in a stable testbed. Furthermore, the performance model proposed in [?] is less comprehensive than the PCM.

In [?], Zheng et al. proposes filters for performance model estimation. The use of performance models is considered limited, since "unknown and time-varying model parameters, such as CPU demands" [?] are difficult to be directly measured. To overcome this issue, the Extended Kalman Filter is applied to estimate hidden parameters indirectly from easily available metrics, such as response time and resource utilization. Therefore, the relation between hidden parameter values and available measurements is modelled as a nonlinear dynamic system. Further on, a systematic methodology for defining a filter estimator is provided. The proposed estimation approach is explored through experiments and considered feasible.

However, parameter values that do not underly Gaussian distributions but, e.g., multi-modal distributions cannot be captured with the proposed approach. This might be critical due to the observations made by Rohr et al.: "Software response time distributions can be of high variance and multi-modal" [?]. Anyhow, developing estimators for performance model parameters is a considerably different approach which requires more studies to answer the question if or under which conditions estimators allow reasonable predictions.

4.2 Application Monitoring at Run-time

Approaches aiming at performance prediction are presented in the previous section. Here in this section, we present existing work concerning general application monitoring at run-time.

4.2.1 Java Applications

In [?, ?] an automatic monitoring framework providing information among different levels involved in the execution is proposed. More specifically, it covers the operation system-, JVM-, middleware- and application-level. It enables tracing the execution and detects poor performance periods according to user-defined performance objectives. Once those objective are defined, "the framework can operate automatically" [?]. The analysis of monitoring data about resource consumption combined with execution behavior primarily aims at hotspot and bottleneck detection.

Systä presents techniques for reverse engineering behavioral models of Java applications [?]. Using static and dynamic analysis, models similar to UML sequence diagrams and UML statechart diagrams are created. Here, event trace information is generated by running the Java application under control of a debugger. Breakpoints allow the extraction of the intra-method control flow. However, running an

application in debugging mode "slows down the execution [...] considerably" [?] and is not feasible for systems in operation. Compared to our target model, the proposed tool set called Shimba does not create any component model and does not consider data flow or data dependencies. UML sequence diagrams are also extracted by [?] and [?] which both use tracing data obtained by instrumentation.

Dynatrace Diagnostics is a tool for performance management that is developed in industry [?]. It traces transactions for applications deployed in distributed, heterogeneous .NET and Java environments. Besides providing a call tree, it also monitors method argument values and provides information about the system's resource utilization [?]. An explicite architecture model is not extracted.

An adaptive monitoring and performance management framework called Compas is proposed in [?, ?]. Performance data is extracted in real-time from a running application and used to generate interaction models describing the system behavior. Contrary to PCM there is no explicit context model defined. "In order to reduce the total overhead of monitoring" [?], an adaptive monitoring concept is proposed that keeps the amount of monitoring small while detecting performance anomalies. It "addresses performance issues related to the EJB layer in J2EE applications" [?]. Components are instrumented by placing a proxy layer around each component. The implementation assumes synchronous invocation style, i.e., it considers session beans and entity beans but no message-driven beans.

In [?, ?], a framework called TestEJB for analyzing J2EE applications is presented. It allows "the measurement of response times, call dependencies inside an assembly, memory consumption of components or single invocations" [?]. Components are instrumented using the interceptor stack provided by the JBoss application server. It remains unclear if the collected response times and memory consumptions could be used for performance prediction. Furthermore, in contrast to our extraction method, the tool does not consider dependencies between parameter values and the control flow.

Kieker [?, ?] is framework to monitor, analyze, and visualize the run-time behavior of Java applications. It does not depend on a special Java execution environment. It can be used to monitor call paths data and response times, and aims at reengineering of, e.g., UML sequence diagrams, Markov chains or timing diagrams. The instrumentation component allows to specify the methods to monitor with Java annotations. In contrast to the WLS-specific WLDF, Kieker does not monitor method arguments. Moreover, WLDF provides information about resource utilization and WLS-specific data which is not possible to obtain with Kieker.

4.2.2 Other Applications

Of course, there are monitoring tools for software other than Java applications. For instance, Tracealyzer is "a viewer and analysis tool for recordings of embedded systems" [?]. Recordings are obtained using separate software recorder that have to be integrated in the base platform of the considered system. It focuses on task scheduling and communication. The system behavior can be presented "at a higher level of abstraction compared to debuggers" [?]. However, the abstraction level is low compared to the architecture model we extract.

4.3 Summary

The related work presented in the previous sections shares several aspects of our approach. Call path tracing using run-time monitoring data, extraction of PCM instances and monitoring resource utilization are not required to be developed from scratch. The novelty is to provide an end-to-end solution for the automated extraction of PCM instances from a running application using state-of-the-art industrial tools.

5. The Extraction Method

In this chapter we describe the method to extract PCM instances from running Java EE applications. Starting with a refinement of the scope in Section 5.1 we present how we intend to build the PCM sub-models. Section 5.8 concludes this chapter by summarizing the method's assumptions and limitations.

5.1 Refining the Scope

In this thesis, we focus on the EJB 3.0 Component Model and do not consider the web tier including Servlets or user interface technologies like Java Server Faces or Java Server Pages. We consider session beans and message-driven beans because they normally encapsulate reusable business logic of Java EE applications. As persistence framework, we use JPA. Traditional entity beans are out of the scope of this thesis and the term "entity" is hereafter used to refer to a JPA entity. As components or elements of components we consider EJBs. JPA entities are not considered as components, since they act as simple data access objects.

We monitor a single WLS instance, clusters of application servers are not in the scope of this thesis. This eases chronological ordering of event records which is needed for call path tracing as presented in Section 3.1.2. For a single WLS instance, the event records are automatically stored in chronological order (see 2.2.1.4). For multiple WLS instances, the server-specific event records would have to be merged and sorted to ensure correct ordering. Sorting the event records in chronological order then would need to handle the issue of clock synchronization in a distributed environment.

The PCM instance extraction of the application under consideration is done semi-automatically. This means that some manual intervention is required. For example, to allow componentization, the component boundaries have to be specified. Additionally, the component code has to be structured in a form that makes intra-component control flow explicit. This information can be provided by the developer or derived from a preceding analysis (see related work in Chapter 4).

We focus on the extraction of the PCM repository and system model. The PCM model of the resource environment is assumed to be specified manually. This is

because the Java EE application server, which is the extractor's main source of information, can not provide complete information on the resource environment by design. For instance, the application server can not know about the properties of the machine the database server is running on. However, the framework presented in [**?**] can extract static information on the resource environment and can therefore be used to provide a PCM resource environment model instance. Furthermore, the PCM allocation model, which depends on the resource environment, is assumed to be manually created. The role of the business domain expert and therefore the PCM usage model is also not considered since end users normally interact with the system through the web tier which, as mentioned earlier, is outside the scope. However, we do consider extracting usage profiles at the component level and at the system boundaries.

5.2 Identifying Component Boundaries and System Boundaries

In our context, a component can be a session bean, a message-driven bean or a set of these EJBs. As stated in Section 3.1.1, we assume information on component boundaries to be available. In order to control the model's granularity, the following options are considered for specifying component boundaries:

- Every EJB is considered as a separate component.
- EJBs contained in the same package are considered as a single component.
- The grouping of EJBs into components is specified explicitly.

Note that we only consider basic components, i.e., we do not consider composite components. A component consisting of more than one EJB is also treated as basic component.

System boundaries are not explicitly declared. The system boundaries are extracted during inter-component control flow extraction. Services that are called from outside, i.e., that are not called from another component and not invoked by a JMS message are marked as system service call. An interface containing marked services is then attached to a system provided role. This method implies that exactly one system is extracted. Furthermore, the extracted system is modelled to only provide and never require services.

The described boundaries allow us to monitor requests both at the system boundary and at the component boundaries. We then can construct probabilistic workload models discussed in detail in Section 5.7.

Table 5.1 shows an exemplary mapping of EJBs to components. `Component1` consists of two EJBs whereas `Component2` and `Component3` each consist of one EJB. We use this as an ongoing example in the following section.

Component name	EJB name
Component1	EJB_A
	EJB_B
Component2	EJB_C
Component3	EJB_D

Table 5.1: Example: EJBs mapped to Components.

5.3 Extracting Inter-Component Control Flow

Extracting the inter-component control flow means obtaining the components and the connections between them. In terms of PCM, parts of the repository model instance and the system model instance are extracted. We use call path tracing in this step to extract the required information. We explain the control flow extraction only for session beans. For message-driven beans listening to messages from JMS queues (asynchronous point-to-point communication) the extraction is similar but requires monitoring of queue-specific message sending and receiving. Publish-subscribe messaging is not considered here.

In order to enable call path tracing, WLDF is configured to instrument the EJB business methods. Each time a business method is reached during processing, we assume two event records to be raised by the injected code: one at the begin of the method and one at the end of the method. Every system request then is represented by a list of event records. According to Section 3.1.2, the list is a call path event record list, i.e., its elements are chronologically ordered.

The example in Figure 5.1 shows a system request's UML sequence diagram. To process a call to method `ClassA#methodA1`, there are four classes involved. At each method entry and method exit, an event record is generated. In the diagram, it is the event record id that is shown. Table 5.2 shows the call path event record list in more detail. The event record ids of the diagram and the table correspond. Each method call is represented by a pair of event records that encloses the event records of inner method calls. The list represents the same behavioral information as the sequence diagram does.

For such a list of event records, inter-component control flow extraction necessitates that the instrumented methods have to be mapped to the components they belong to. This additional information then allows for extraction of the control flow between components.

As presented in Table 2.1, an event record provides fully qualified names about the instrumented method and its surrounding class. First we have to figure out which EJB's business method the instrumented method implements. In other words, the instrumented method has to be mapped to an EJB. Normally, an EJB is implemented by exactly one implementation class, but this is not a requirement. An EJB may be implemented by more than one class, and one class may implement more than one EJB. In our context, we assume that we can unambiguously map a method together with its surrounding class to an EJB. Once the EJB is determined, it can be mapped to its containing component as described in the preceding section. The instrumented business method is then interpreted as a service in terms of PCM.

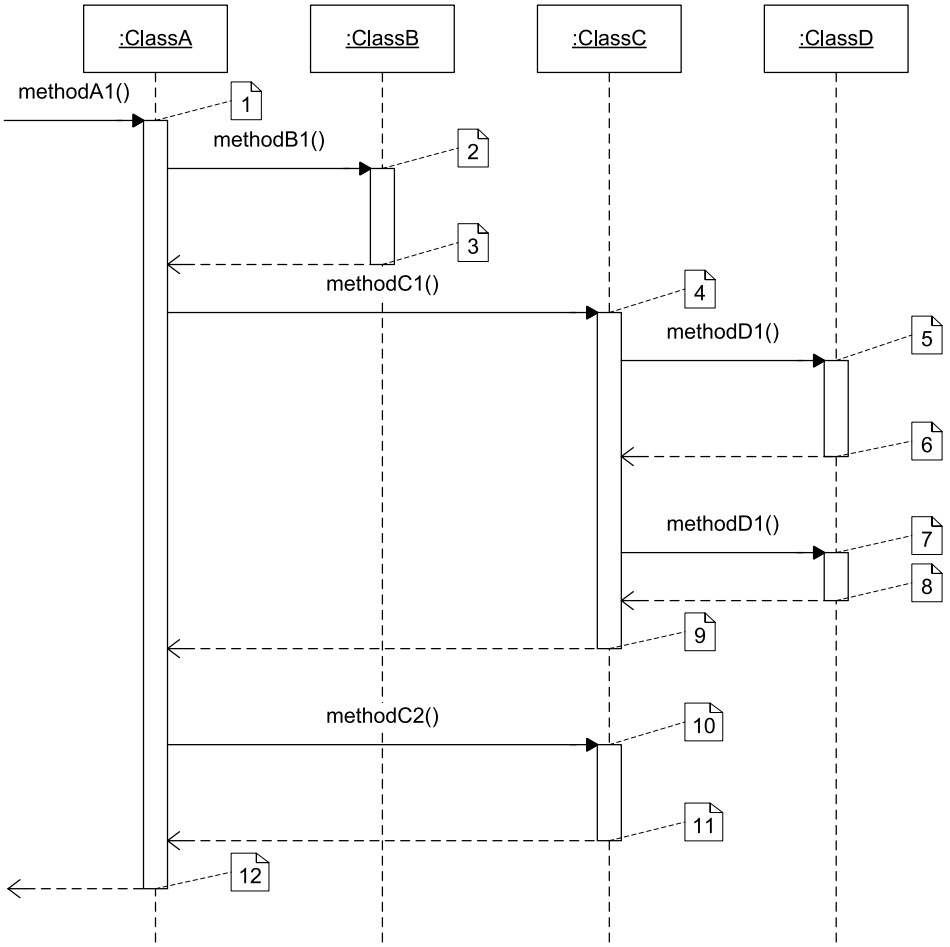


Figure 5.1: Example: UML sequence diagram showing a request.

Id	**Class name**	**Method name**	**Location** (method entry/exit)
1	`Class_A`	`methodA1`	entry
2	`Class_B`	`methodB1`	entry
3	`Class_B`	`methodB1`	exit
4	`Class_C`	`methodC1`	entry
5	`Class_D`	`methodD1`	entry
6	`Class_D`	`methodD1`	exit
7	`Class_D`	`methodD1`	entry
8	`Class_D`	`methodD1`	exit
9	`Class_C`	`methodC1`	exit
10	`Class_C`	`methodC2`	entry
11	`Class_C`	`methodC2`	exit
12	`Class_A`	`methodA1`	exit

Table 5.2: Example: Call path event record list.

EJB name	Implementation class
EJB_A	ClassA
EJB_B	ClassB
EJB_C	ClassC
EJB_D	ClassD

Table 5.3: Example: Mapping of EJBs and implementation classes.

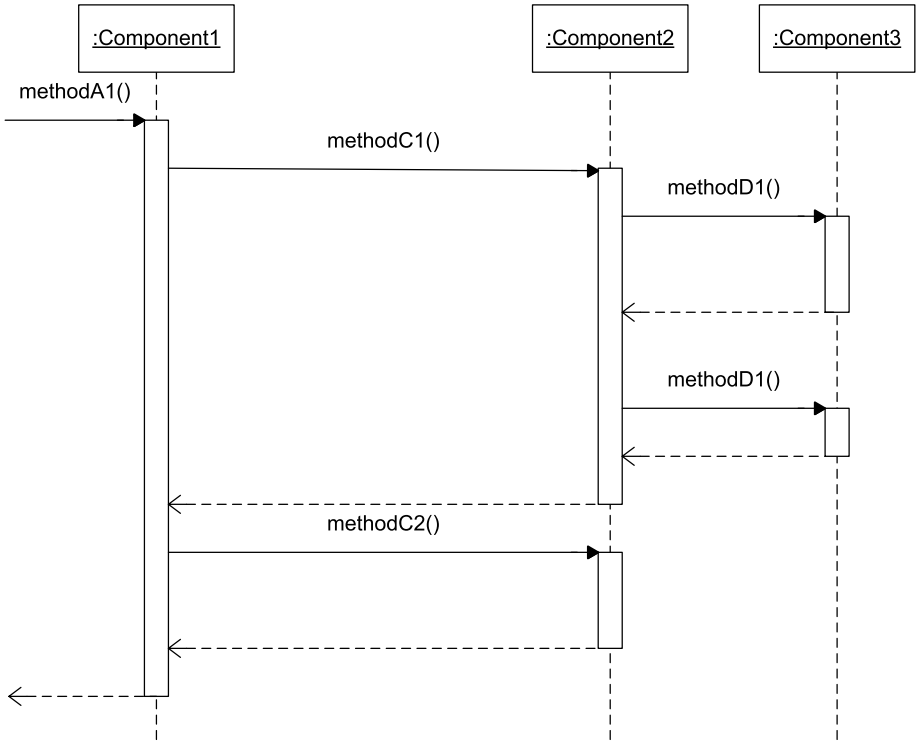


Figure 5.2: Example: UML sequence diagram showing a request (to components).

In the ongoing example, for reasons of clarity we assume a simple mapping of EJBs to their implementation classes, i.e., each EJB is implemented by exactly one class. In Table 5.3, the mapping is depicted. Figure 5.2 shows the same system request as Figure 5.1 but relates the request to the corresponding components. More precisely, the sequence diagram shows component instances. The main difference between the two UML sequence diagrams is that classes `ClassA` and `ClassB` are embraced by component `Component1`.

The inter-component control flow is then used to build a PCM repository model and a PCM system model.

- **PCM repository model.** Whereas the set of components is already fixed by the mapping of EJBs to components, the interfaces and provided/required roles are inferred from the observed service calls between the components. Considering a service call from `ComponentX` to `ComponentY`, the service call is registered as a provided service of `ComponentY` and a required service of `ComponentX`. The provided services of one component are summarized in one interface that is then attached to the component via provided role. This in-

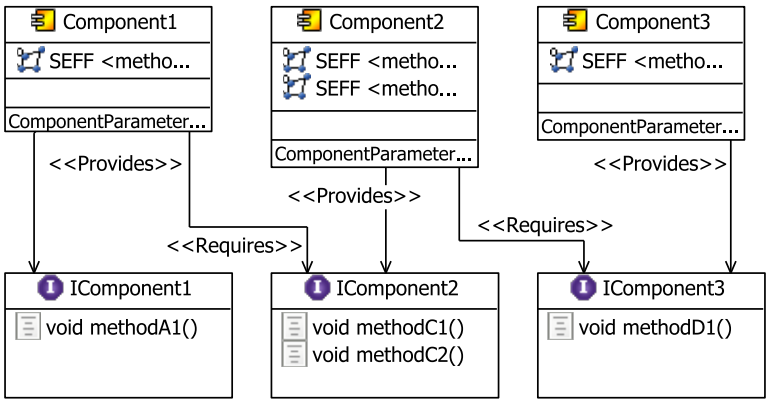


Figure 5.3: Example: Repository model.

terface is also attached to those components that require a service contained in the interface, but then via required role. Thus, a component's provided i nterface consists only of those provided services that are obtained during the dynamic analysis. The same applies for the component's required interface. Hence, the provided interface does not necessarily correspond to the set of Java interfaces implemented by the component's EJBs.

- **PCM system model.** The observed connections between the components are directly used to build a system model. For each invoked component, an assembly context is created. A service call between two components results in an assembly connector whereas a service call from "outside", i.e., a call to a component without a successor in the call path event record list, results in a delegation connector and a corresponding system provided interface. This does not apply for calls to MDBs. A call to an MDB has no successors in the call path event record list, but it should not result in a system provided service.

Figures 5.3 and 5.4 show the repository model respectively the system model based on the information given in Table 5.2. Further system requests result in extending the models if not already captured methods are called or if methods are called from a not already registered caller.

For reasons of simplicity, method arguments are omitted in the example. However, modelling interfaces and their services requires modelling of method parameters. Therefore, data types have to be mapped to PCM entities as well.

The inter-component control flow extraction produces a PCM system model and a PCM repository model. While the latter already contains components and interfaces, the internal structure of components has to be extracted in a further step.

5.4 Extracting Intra-Component Control Flow and Parametric Dependencies

A Palladio component specification includes information on the component-internals. To capture the performance-relevant behavior of provided services using RDSEFFs,

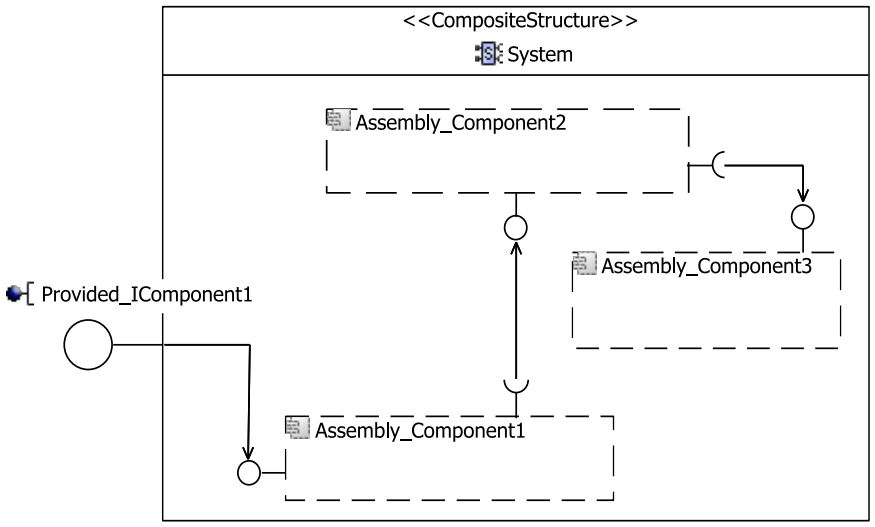


Figure 5.4: Example: System model.

the intra-component control flow must be modelled. In the following we examine the extraction of the control flow in detail. In the following we examine the extraction of the control flow in detail.

In order to abstractly describe the control flow of a component service, it is assumed that performance-relevant code fragments are labeled. To make monitoring of those fragments possible, the component code has to be structured in a form that makes performance-relevant intra-component control flow information explicit. This requirement arises from the lack of tool support for in-method instrumentation. Current tools do not support instrumentation at defined locations other than method entries/exits or method calls [**?**].

We use a schema for organizing component code in such a way that RDSEFF information about sequences of PCM `AbstractActions` is encoded in the structure. In other words, we refactor component implementations by moving performance-relevant actions to separate methods named according to a defined naming schema. This enables WLDF measurements regarding, e.g., parameter dependencies or CPU resource demands.

We first present general information about the *extract method* refactoring. Then in Sections 5.4.1 and 5.4.2 we introduce the naming schema and describe how it is used to help extracting intra-component control flow and parametric dependencies.

Refactoring: Extract Method.

Code refactoring means changing code structure without modifying the code's functional behavior. The aim is to improve extra-functional properties of the code like, e.g., maintainability or extensibility [**?**, **?**]. In our context, refactoring is employed to allow the monitoring of method internals. The refactoring we use is the extract method refactoring. Figure 5.5 shows a basic example: The code fragment doing some heavy computation inside method `computeSomething` is moved into a separate method.

If the extracted code fragment contains references to local variables that are defined outside the extracted code fragment, these references have to be passed as method

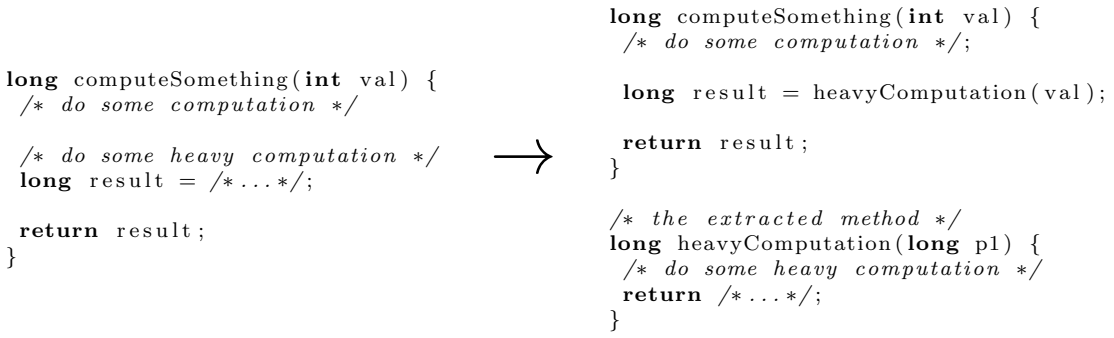


Figure 5.5: Refactoring: Extract method.

`<methodname>`	=	`<pname>` '_' `<idA>` '_' `<action>` '_' `<idZ>`
`<pname>`	=	The name of the parent method.
`<action>`	=	'internalaction' | 'externalcallaction' | 'loopaction' | 'loopbody' | 'branchaction' | 'branchtransition'
`<idA>`	=	A number that, together with `<pname>`, uniquely identifies the parent method. In case of method overloading, the parent method's name alone could not be used as identifier.
`<idZ>`	=	A number that uniquely identifies the extracted method amongst those extracted methods with the same parent method. Note that the number only serves as identifier, it is not an index of a sequence.

Figure 5.6: Naming of extracted methods representing performance-relevant actions.

arguments. If the extracted code fragment contains assignments to local variables that are referred after the extracted code fragment, the assigned values have to be returned by the method. Given that Java supports only one return value per method, a helper class encapsulating the return values has to be introduced in case there are multiple values to return.

5.4.1 Intra-Component Control Flow

We first present the naming schema for the relevant PCM actions and then describe how we extract the intra-component control flow from monitoring data.

Naming Schema

In order to make monitoring of performance-relevant control flow possible, a method implementing a service that is provided by a component has to be refactored. Performance-relevant actions are extracted to separate, newly created methods. We refer to the method from which the newly created methods are extracted as the *parent method*. The names of the extracted methods are used to encode information about the parent method's control flow. The naming schema is introduced in Figure 5.6.

The schema is intended for the PCM actions *internal action*, *external service call action*, *loop action* and *branch action*. PCM fork actions are not considered in our context, since we refactor only EJB implementations which are required to be single-threaded and cannot spawn new threads. Figures 5.7, 5.8 and 5.9 illustrate the respective refactorings for internal actions, external call actions, loop actions and branch actions. As method to refactor, we consider method `mymethod`. The expressions `<FormalParameterList>`, `<ActualParameterList>` and `<ReturnType>` are obtained as described in the preceding section. The listings depict how the respective actions have to be moved to separate methods. The listings are not intended to explain how performance-relevant actions could be identified. As already mentioned, we assume this information to be available. It can be provided by the developer or derived from a preceding analysis (see related work in Chapter 4).

In our context, refactoring the code of a method of interest might require method inlining. Method inlining means to replace a method call by the code of the method being called. More precisely, a method that is called by the method of interest needs to be inlined i) if it is not an external service and ii) if the method itself or one of its successors contains a call to an external service. Once the performance-relevant actions have been identified and the process of method inlining has been finished, the refactoring can be carried out. Note that the identification of performance-relevant actions depends on the component boundaries. The component boundaries determine whether a service call is an external service call or not.

Control Flow Extraction

From now on, we assume that the component services have been refactored according to the described naming schema. The intra-component control flow is extracted by means of call path tracing similar to the process described in Section 5.3: WLDF is configured to instrument, in addition to the EJB business methods, the newly created methods. Each time such a method is reached during processing, one event record at the method entry and one event record at the method exit is generated by the injected code. Thus, for a single system request we can obtain a call path event record list containing in addition to the event records for EJB business methods also event records for the newly introduced methods representing performance-relevant actions. Using the information encoded in the method names, intra-component control flow can be extracted. The RDSEFF in Figure 2.6 shows a control flow that has been extracted using such monitoring data.

In addition to the control flow, we can also extract branching probabilities and probability mass functions (PMFs) for loop iteration numbers. By dividing a performance-relevant loop into a loop action method and a loop body method, for each service execution the loop iteration number can be obtained. Aggregating data from multiple executions then leads to a PMF for the loop iteration number. This way, the loop iteration number as shown in loop action `loopAct3` in Figure 2.6 can be estimated. Similar to loop iteration numbers, because of the separation of branch action methods and branch transition methods, branch transition probabilities can be estimated. A loop iteration number specification becomes lengthy if the set of different observations is large. In this case, the PMF can be made more coarse-grained by merging the probabilities for loop iteration number intervals.

Furthermore, since the internal actions now reside in separate methods, measuring the response times of internal actions is possible.

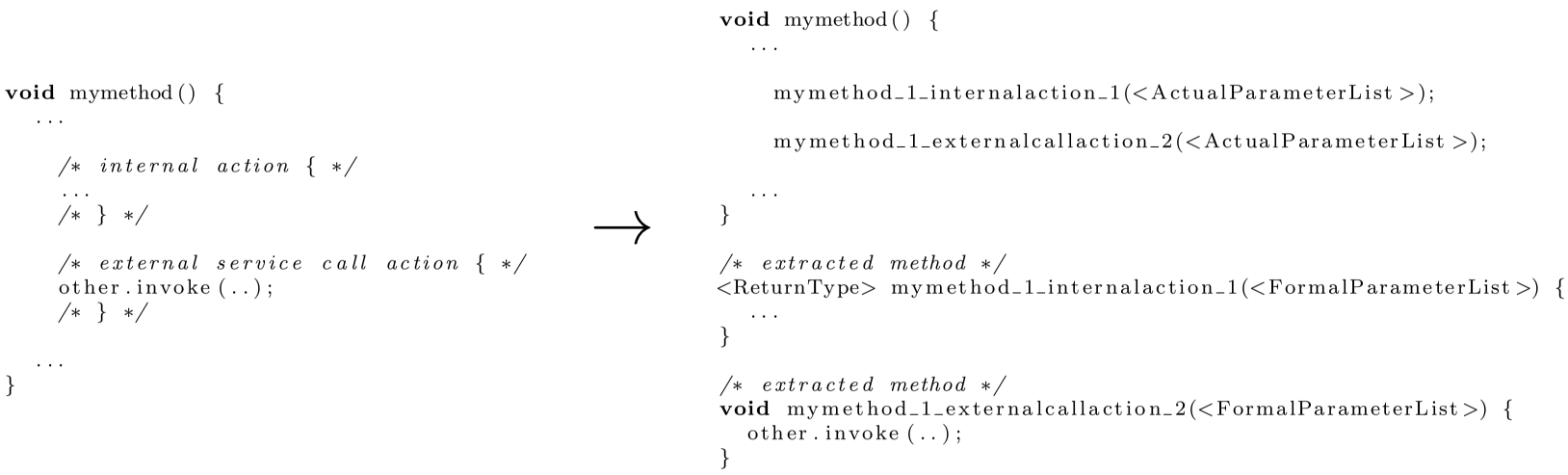


Figure 5.7: Extracting an internal action and an external service call action.

```
void mymethod() {
  ...
    /* loop action { */
    while(..) {
      /* loop body { */
      ...
      /* } */
    }
    /* } */
  ...
}
```

```
void mymethod() {
  ...
    mymethod_1_loopaction_1(<ActualParameterList>);
  ...
}

/* extracted method */
<ReturnType> mymethod_1_loopaction_1(<FormalParameterList>) {
  while(..) {
    mymethod_1_loopbody_2(<ActualParameterList2>);
  }
}

/* extracted method */
<ReturnType> mymethod_1_loopbody_2(<FormalParameterList>) {
  ...
}
```

Figure 5.8: Extracting a loop action.

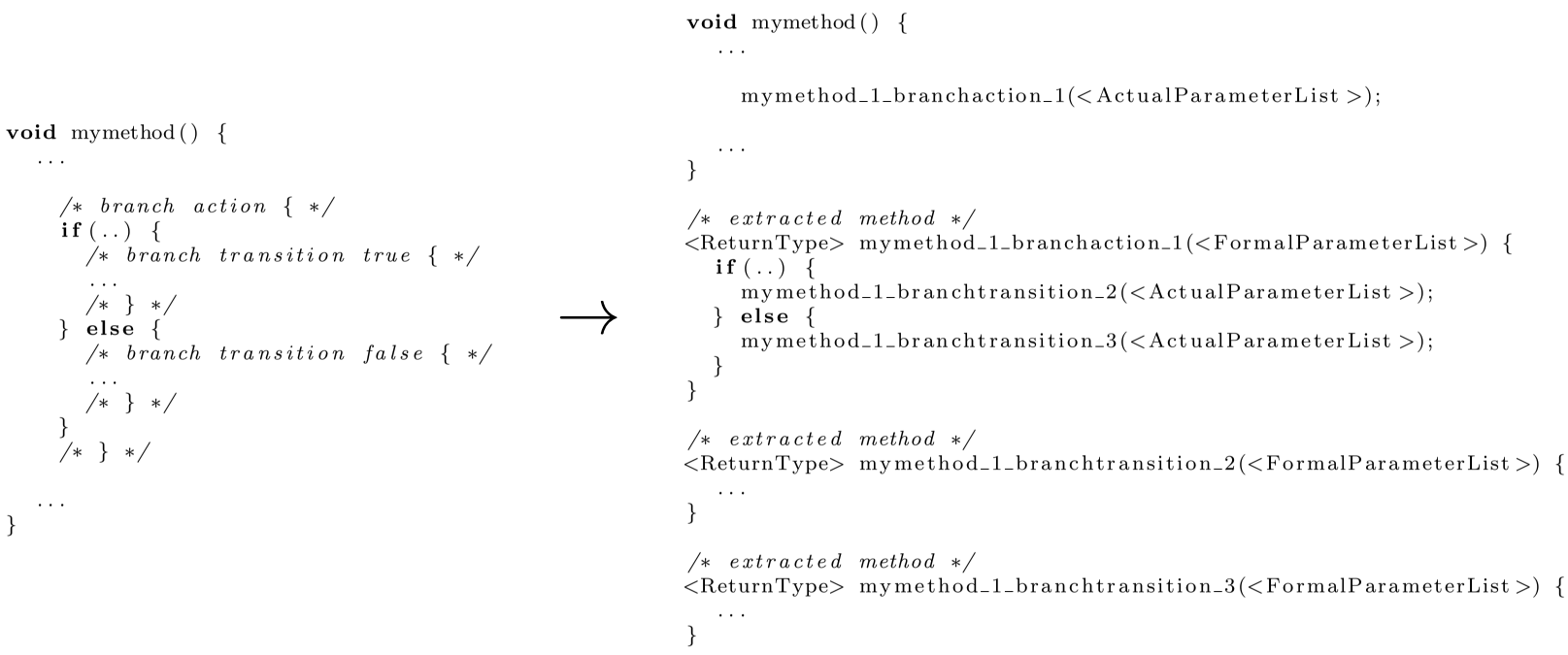


Figure 5.9: Extracting a branch action.

5.4.2 Parametric Dependencies

Using the procedure described in the previous section, branching probabilities and statistics for loop iteration numbers can be collected. However, one of PCM's outstanding strengths is the support for extensive parameterization. In PCM, one "can specify the dependencies between input parameters and resource demands, branch probabilities, or loop iteration numbers in RDSEFFs" [**?**].

In contrast to techniques using static code analysis we cannot extract explicit parametric dependencies from monitoring data. Instead, we can monitor method parameter and relate observed parameter values to the observed method control flow. As an example, for each observed parameter value we can monitor which branch of an if-statement is taken or how often a loop body is executed. Thus, based on monitoring data probabilistic models of parametric dependencies can be extracted. This way, it is even possible to observe parametric dependencies that would not be covered by static code analysis.

Automatically detecting relevant parametric dependencies using only monitoring data requires, e.g., an analysis of correlations or a machine learning technique (see [**?**]). If no previous knowledge is available, an automatic detection method has to cope with a high degree of freedom: For each action the method has to figure out to what extent the action depends on available parameters. Furthermore, parametric dependencies are not restricted to parameter values, PCM supports parameterization for multiple *characterization types* like, e.g., a parameter's byte size. We assume that potential performance-relevant parametric dependencies are already indicated. Such information might stem from the developer, from an automatic detection method as described above, from static code analysis, or other analysis methods. With the knowledge of which parametric dependency to observe we then use monitoring data to probabilistically quantify that dependency. Thus, we only consider those dependencies that already marked as performance-relevant.

In analogy to the previous section, performance-relevant parametric dependencies are encoded in method names using a specific naming schema for the extracted methods.

Extended Naming Schema

The naming schema we introduce to encode performance-relevant parametric dependencies is an extension of the schema presented in Figure 5.6. This means, the parameter dependency annotations are attached to the methods where the performance-relevant actions are moved to (see Section 5.4.1). Figure 5.10 shows the syntax of the extended schema. For the definition of the gray-colored non-terminal symbols see Figure 5.6. Parameter dependency declarations (`<pddeclaration>`) are only allowed for methods representing branch actions or loop actions. We do not consider dependencies between input parameters and resource demands of internal actions. We discuss the extraction of resource demands in Section 5.6.

In PCM, actions can depend on multiple parameters and on multiple characterization types. The extended naming schema also allows the declaration of action-specific parametric dependencies. A parameter dependency declaration consists of a parameter reference and a characterization type. In PCM, a parameter can be a method input parameter or a local variable. A local variable may be initialized as return

`<methodname>`	=	`<pname> '_' <idA> '_' <action> <ext> '_' <idZ>`
`<ext>`	=	`<pddeclaration> | <pddeclaration> <ext> |`
`<pddeclaration>`	=	`'_pardep_' <parref> '_' <varchartype>`
`<parref>`	=	`<paramindex>`
`<paramindex>`	=	The index of the parent method's input parameter list.
`<varchartype>`	=	A PCM variable characterization type represented as String, e.g., `'VALUE'` or `'NUMBER_OF_ELEMENTS'`.

Figure 5.10: Extended naming schema to annotate parameter dependencies.

value of a preceding external service call. The extended naming schema allows referencing only input parameters. The input parameter is identified by its index in the parent method's parameter list. This is because WLDF does not work with parameter names but with parameter lists.

Figure 5.11 illustrates exemplary refactorings to encode identified performance-relevant parametric dependencies into the extracted method names. The extracted methods stem from refactorings conducted in the previous step as described in Section 5.4.1. The branch action of method `mymethod` is considered to depend on the value of the method's input parameter `a`. The loop action is considered to depend on the value of the method's input parameter `b`.

We do not model data flow, i.e., explicit propagation of parameter characterizations. Hence, encoding performance-relevant parametric dependencies is only appropriate for implementations of services that are exposed to the system boundary (for the same reason we do not model the setting of return variables in a RDSEFF). Let us assume we would extract a parametric dependency for a service that is not exposed to the system boundary, e.g., a dependency between the value of an input parameter `x` and a loop iteration number. To conduct simulations the value of `x` would then have to be set. However, we do not model how the input parameters of an external service call are set. Therefore, we would have to extract how parameters characterizations are propagated. An automated extraction based on monitoring data would require, e.g., an analysis of correlation between input parameters and passed parameters or an application of a machine learning technique [**?**]. Given that we do not provide such techniques we can extract parametric dependencies only at the system boundary. Only those parameters that occur at the system boundary can be explicitly specified, namely via PCM usage models.

Nevertheless, note that the extraction of parametric dependencies also works for components that are not directly accessed through a system interface. We just cannot simulate the respective RDSEFFs in the context of the models we extract, but in a (manually) adapted model the extracted parametric dependencies might be useful.

Dependency Extraction

The inner control flow of a component service can be extracted as proposed in Section 5.4.1. WLDF is configured to instrument, in addition to EJB business methods, extracted methods representing performance-relevant actions. More precisely, method entries and method exits are instrumented. If WLDF is configured to inject

```
void mymethod(int a, int b) {
  ...

    /* external service call action { */
    int result = other.invoke(..);
    /* } */

    /* branch action depending on the value
     * of parameter a { */
    if(..) {
      /* branch transition true { */
      ...
      /* } */
    } else {
      /* branch transition false { */
      ...
      /* } */
    }
    /* } */

    /* loop action depending on the value
     * of parameter b { */
    while(..) {
      /* loop body { */
      ...
      /* } */
    }
    /* } */

  ...
}
```

→

```
void mymethod(int a, int b) {
  ...

    int result = mymethod_1_externalcallaction_1(<ActualParameterList>);
    mymethod_1_branchaction_pardep_0_VALUE_2(<ActualParameterList>);
    mymethod_1_loopaction_pardep_1_VALUE_5(<ActualParameterList>);

  ...
}

/* extracted method */
int mymethod_1_externalcallaction_1(<FormalParameterList>) {
  return other.invoke();
}

/* extracted method */
<ReturnType> mymethod_1_branchaction_pardep_0_VALUE_2(<FormalParameterList>) {
  if(..) {
    mymethod_1_branchtransition_3(<ActualParameterList>);
  } else {
    mymethod_1_branchtransition_4(<ActualParameterList>);
  }
}
/* extracted method */
<ReturnType> mymethod_1_branchtransition_3(<FormalParameterList>) {
  ...
}
/* extracted method */
<ReturnType> mymethod_1_branchtransition_4(<FormalParameterList>) {
  ...
}

/* extracted method */
<ReturnType> mymethod_1_loopaction_pardep_1_VALUE_5(<FormalParameterList>) {
  mymethod_1_loopbody_6(<ActualParameterList2>);
}
/* extracted method */
<ReturnType> mymethod_1_loopbody_6(<FormalParameterList>) {
  ...
}
```

Figure 5.11: Example: Indicating performance-relevant parametric dependencies.

a `DisplayArgumentsAction` at the EJB business method's entry, the corresponding event records contain information about the method's input parameter. This way, parameters can be monitored and related to the actual control flow.

In its current version, WLDF provides only String representations of the monitored parameters. The String representation is derived from the `toString` method of class `java.lang.Object`. For parameters of primitive type, their String representations are derived from the `toString` method of the corresponding Java primitive wrapper classes. This is why we cannot monitor all parameter types and all PCM variable characterization types. For instance, the value of a parameter of type `int` can be observed while the byte size of the parameter cannot be observed.

To discuss how specified parameter dependencies can be quantified, in the following we observe a single component service. Thanks to the annotated parameter dependencies it is known which parameter variable characterization should be related to which performance-relevant actions. For reasons of simplicity, we assume that the actions depend on at most one input parameter and at most one PCM variable characterization type. Then for each service execution and for each parameter dependent action a tuple can be obtained that consists of a parameter characterization and an information about which control flow path has been taken.

- For a branch action with $bt_1, \ldots, bt_m$ branch transitions, the monitored control flow path is characterized by an index $i, 1 \leq i \leq m$.
- For a loop action, the monitored control flow path is characterized by the observed loop iteration number $i, i \geq 0$.

Let v be an observed parameter characterization, then $t = (v, i)$ is such an observed tuple for one parameter dependent action. Considering multiple service executions, then for each parameter dependent action a there is a list $L_a = (t_{a,1}, \ldots, t_{a,k})$. In the following, let the action a be fixed so that the index a can be omitted, i.e., $L_a = L = (t_1, \ldots, t_k)$. From the observed tuple list, PMFs can be derived. Let $V = \{v_k | t_k = (v_k, i_k) \in L\}$ be the set of observed parameter characterizations. Let $C = \{i_k | t_k = (v_k, i_k) \in L\}$ be the set of observed control flow paths. Then for each $v \in V$, there is a PMF $f_v : C \to [0, 1]$. The PMF f_v is defined as

$$f_v : C \to [0,1], \quad f_v(i) = \frac{\#\{t_k | t_k = (v_k, i_k) \in L \wedge v_k = v \wedge i_k = i\}}{\#\{t_k | t_k = (v_k, i_k) \in L \wedge v_k = v\}} \ .$$

Informally, $f_v(i)$ is the number of observed tuples (v, i) divided by the number of observed tuples where v is involved. Notice that the PMF evaluates to zero, i.e., $f_v(i) = 0$, if $i \notin C_v = \{i_k | t_k = (v_k, i_k) \in L \wedge v_k = v\}$ where C_v is the set of observed control flow paths for v.

Thus, we quantify parametric dependencies by PMFs. The question now is how these PMFs can be integrated in the PCM model.

- For a branch action, there are PMFs $f_v : C_v \to [0, 1]$ with C_v the observed subset of $1, \ldots, m$ (m is the number of possible branch transitions). Figure 5.12 sketches an example of how these PMFs can be represented in PCM. There is a branch action that depends on the parameter characterization `x.VALUE`. There

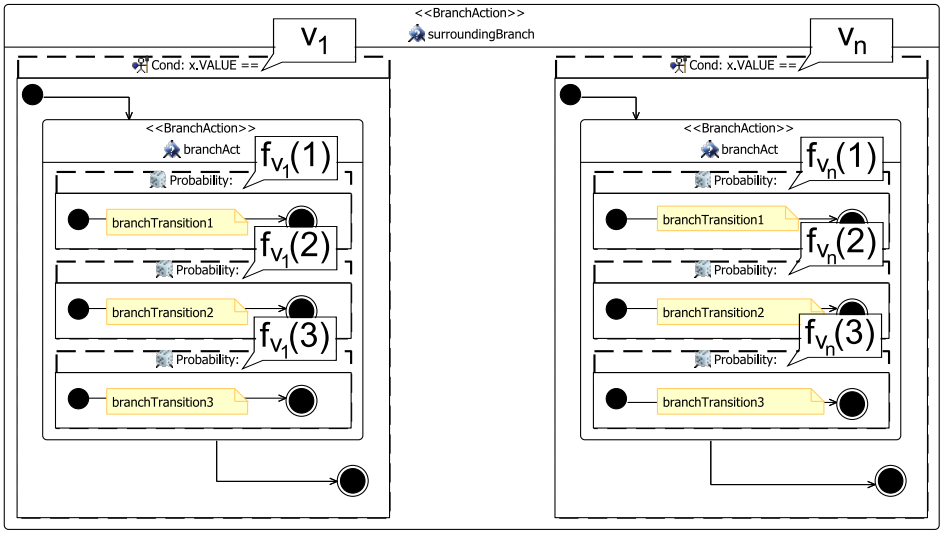


Figure 5.12: Example: Exemplary branch action depending on `x.VALUE`.

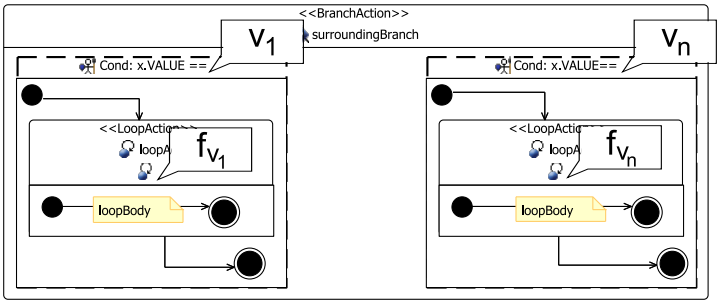


Figure 5.13: Example: Exemplary loop action depending on `x.VALUE`.

are three probabilistic branch transitions for the branch actions, i.e., $m = 3$. Instead of providing a single probability for a branch transition the transition probabilities depend on `x.VALUE`. A surrounding branch action is added. For each $v \in V = \{v_1, \ldots, v_n\}$, a guarded branch transition is introduced.

- For a loop action, there are PMFs $f_v : C_v \to [0, 1]$ with C_v a subset of the observed loop iteration numbers. Figure 5.13 shows an example of how the PMFs can be represented in PCM. There is a loop action that depends on the parameter characterization `x.VALUE`. There is no single PMF for the loop iteration number, the loop iteration number depends on `x.VALUE`. A surrounding branch action is added. For each $v \in V = \{v_1, \ldots, v_n\}$, a guarded branch transition is introduced.

Note that the parameter characterization dependent branch transition probabilities or loop iteration numbers can be also modelled using the conditional operator "? : ". In Figures 5.12 and 5.13, for reasons of clarity, we chose a surrounding branch action to model the different cases.

If the set $V = \{v_1, \ldots, v_n\}$ is large, distinguishing the branch transition probabilities respectively loop iteration numbers for each v_i is impractical. In this case, we propose to partition the set V and work with partitions instead of individual

values. The branch transition probabilities respectively loop iteration numbers can be specified partition-wise. The PMF representing one partition can be computed by aggregation: Let $P_r = \{v_{r,1}, \ldots, v_{r,l}\} \subseteq V$ be such a partition. Then the PMF $f_r : C \to [0,1]$ representing the partition P_r is defined as the normalized weighted sum of the PMFs $f_{v_{r,1}}, \ldots, f_{v_{r,l}}$:

$$f_r : C \to [0,1], \quad f_r(i) = \frac{\sum_{j=1}^{l} p(v_{r,j}) f_{v_{r,j}}(i)}{\sum_{j=1}^{l} p(v_{r,j})}$$

with

$$p : C \to [0,1], \quad p(v) = \frac{\#\{t_k | t_k = (v_k, i_k) \in L \wedge v_k = v\}}{\#V}$$

responsible for weighting and normalization.

5.5 Obtaining Information on the Resource Environment

The PCM distinguishes two types of resources. There is a passive resource type with semaphore semantics and a processing resource type that is characterized by a processing rate. A specialization of the latter, a linking resource type, is used to represent network resources.

We assume a PCM resource environment model to be available. A PCM resource environment consists of resource containers and possibly linking resources connecting them. A resource container then contains processing resources. The execution environment of an application server including a database tier is complex. Due to the reasons given in 5.1, we consider a single WLS instance. In addition, a model of the database server (DBS) has to be provided.

Passive resources are not modelled within a PCM resource environment model. They are modelled within the PCM repository model, attached to components. The components we introduced in Section 5.2 group EJBs in an architectural sense. At the component level, EJB instance pools can be modelled as passive resources. In order to model passive resources that belong to the Java EE container level, a separate component has to be introduced. Database connection pools and the WLS global thread pool are such resources. They reside at the container level and can be modelled as passive resources. Respective pool sizes can be obtained by reading the WLS settings from Runtime MBeans.

5.6 Extracting Resource Demands

Resource demands include demands for passive resources and demands for processing resources as they are modelled in the PCM repository respectively resource environment model. Obviously, modelling resources and extracting resource demands have to be aligned. Thus, this section builds on the previous section.

5.6.1 Demands for Passive Resources

In PCM, a demand for a passive resource consists of two actions: *acquire resource* action and *release resource* action. Passive resources considered in Section 5.5 model, e.g., EJB instance pools, JDBC connection pools or the WLS global thread pool.

The usage of an EJB instance pool is implicitly given when a request calls a service that is provided by an EJB. Within a WLS instance, a typical request is processed by one thread. Since we consider only one WLS instance, one request then would trigger two WLS global thread pool accesses. At the beginning of the request, the pool is asked for thread acquisition. At the end of the request, the acquired thread is released. One possibility to model this behavior is to provide a wrapper component for all system requests. Such a component then has to provide all interfaces that are provided by the system, the RDSEFFs have to delegate the services with surrounding acquire and release actions. However, in general, a request is not pinned to one specific thread, the processing thread may vary. Thus, an exact modelling of acquire and release actions is not possible.

Additional information is required to find out when the JDBC connection pool is accessed. At first, one has to find out which services make use of JDBC connections. However, even if this information is available, exact modelling of acquire and release actions is difficult. This is because a request normally comprises multiple service calls and instead of acquiring and releasing a JDBC connection each time the DBS is used, WLS makes optimizations. Within a WLS instance, the WLS may pin the JDBC connection to a request the first time the connection is used. Further service calls then make use of the same JDBC connection. This means that the position of the acquire action is not fixed. It depends on the concrete sequence of service calls. Again, an exact modelling of acquire and release actions is impractical.

5.6.2 Demands for Processing Resources

Unlike single demands for passive resources, demands for processing resources have to be quantified. It is not sufficient to model which action requires specific resources, one also has to provide information to what extent the resources are stressed. In PCM, resource demands for processing resources are attached to internal actions.

With reference to Section 5.5, we have to extract the Java EE application's demands for resources underlying the WLS instance, the DBS and the connections between them. Demands for memory resources like physical memory or Java heap memory are outside the scope of the thesis. Furthermore, we do not consider disk usage or data transfer over the network. We do consider CPU resources on the application server and CPU resources on the DBS.

In the subsequent sections we discuss how total resource demands can be distributed between the application server CPU and DBS CPU and how demands for the specific resources can be quantified.

5.6.2.1 Apportioning Demands among Different Resources

We assume a typical EJB business method to perform the following activities in arbitrary order in the context of a transaction: Reading some data from a database, computing something and writing some data to a database. Note that database

```
class LargeOrderSession {
    ...
    public LargeOrder createLargeOrder(String assemblyId, ...) {
      // extracted external call
      Assembly assembly = createLargeOrder_1_externalcallaction_1(assemblyId);
      // extracted internal action
      LargeOrder largeOrder = createLargeOrder_1_internalaction_2(assembly, ...);
      return largeOrder;
    }

    private LargeOrder createLargeOrder_1_internalaction_2(Assembly assembly, ...)
        {
      /* compute something */
      LargeOrder largeOrder = new LargeOrder(assembly, ...);
      entityManager.persist(largeOrder);
      return largeOrder;
    }

    private Assembly createLargeOrder_1_externalcallaction_1(String assemblyId) {
      Assembly assembly = mfgSession.findAssembly(assemblyId);
      return assembly;
    }
}

class MfgSession {
    ...
    public Assembly findAssembly(String assemblyId) {
      // extracted internal action
      return findAssembly_1_internalaction_1(assemblyId);
    }

    private Assembly findAssembly_1_internalaction_1(String assemblyId) {
      return entityManager.find(Assembly.class, assemblyId);
    }
}
```

Figure 5.14: Example: EJB business method `createLargeOrder`.

accesses are typically not directly performed by EJB business method code, they are encapsulated in JPA entity accesses. This information hiding is helpful for EJB developers, but complicates the resource demand extraction.

WLDF provides system-level monitors that can be configured to record database activities. Diagnostic actions attached to monitors `JDBC_Before_Statement_Internal` and `JDBC_After_Statement_Internal` are triggered whenever a JDBC statement is executed. With the help of the diagnostic context id, the JDBC statements can be related to a specific request. Figure 5.14 shows an EJB business method implementation. The method `createLargeOrder` in class `LargeOrderSession` is refactored according to the schema proposed in Section 5.4.1. It consists of an external service call and an internal action. The called external service is the business method `findAssembly` of another EJB, the method's implementation consists of a single internal action. Focusing on JPA usage, the internal action of method `createLargeOrder` persists a new instance of entity `LargeOrder` whereas the internal action of called service `findAssembly` tries to find an instance of entity `Assembly`.

For the exemplary call path event record list shown in Table 5.4, we assume WLDF to be configured to instrument EJB business methods with the predefined monitors `EJB_Before_SessionEjbBusinessMethods` and `EJB_After_SessionEjbBusinessMethods`, to instrument methods representing performance-relevant actions with custom monitors `CustomMonitor_Before_Action` and `CustomMonitor_After_Action`

and to instrument execution of JDBC statements with the predefined monitors `JDBC_Before_Statement_Internal` and `JDBC_After_Statement_Internal`. Let the event record list be triggered by a request for `createLargeOrder`. The example is intended to illustrate the issues involved in separating demands for the application server and the DBS. Note that the search for an instance of entity `Assembly` leads to a JDBC access performed within the surrounding internal action (`Id` 5 and 6). In contrast, writing of an instance of entity `LargeOrder` does not take place within the surrounding internal action. The JDBC access is performed after method `createLargeOrder` is processed (`Id` 13 and 14). This is because the request is processed inside a transaction. The database write is conducted during the commit phase of the transaction. In this case, the commit phase follows the processing of method `createLargeOrder`. Hence, we can distinguish between database read and write if we observe when a JDBC statement is processed.

Measuring response times of JDBC statements is not possible with the current WLDF version. There is no JDBC statement monitor that is compatible with the time measuring diagnostic actions `TraceElapsedTimeAction` or `MethodInvocationStatisticsAction`. The latter action cannot be applied for another reason: That action does not create event records, but aggregates measurement data to provide statistics. Thus, there is no context id and the measurements cannot be related to a specific service call.

In order to apportion resource demands among the application server and the DBS, we propose an approximation. With regard to a transaction, we split the resource demands at the boundary between working phase and commit phase. The working phase is assumed to stress the application server, the commit phase is assumed to stress the DBS. This approximation introduces several errors: i) The resource demands of database writes are overestimated. Database writes are part of the commit phase, but the commit phase also produces overhead on the application server, e.g., overhead for managing the transaction which is different for a one-phase or a two-phase commit. ii) The resource demands of database reads are ignored. iii) The resource demands of the application server are either over- or underestimated, depending on whether the database access including reads and writes is under- or overestimated. The accuracy of the approximation depends on the characteristics of the transaction. If the transaction performs only database reads and no writes, the error will be higher than for a transaction whose database reads and writes are balanced. Note that a read on a JPA entity does not always entail a database read because JPA implementations provide data caching. Furthermore, the approximation abstracts from the location where database accesses take place. Database writes cannot be assigned to the methods where they are actually triggered. Resource demands for the application server we encode in the respective internal actions. The resource demands for the DBS we delegate to a specific database access interface we introduce together with a providing component. Later, in the case study in Section 7.4, the approximation is tested for adequacy.

Applying the approximation requires knowledge about the transaction boundaries. An EJB business method may run inside a transaction, outside a transaction or partly inside a transaction. The EJB 3.0 standard supports bean-managed transactions and container-managed transactions. The former concept allows the EJB developer to designate transaction boundaries. In the latter concept, it is the EJB

Id	Monitor name	Class name	Method name
1	EJB_Before_SessionEjbBusinessMethods	LargeOrderSession	createLargeOrder
2	CustomMonitor_Before_Action	LargeOrderSession	createLargeOrder_1_externalcallaction_1
3	EJB_Before_SessionEjbBusinessMethods	MfgSession	findAssembly
4	CustomMonitor_Before_Action	MfgSession	findAssembly_1_internalaction_1
5	JDBC_Before_Statement_Internal	JDBCConnectionWrapper	prepareStatement
6	JDBC_After_Statement_Internal	JDBCConnectionWrapper	prepareStatement
7	CustomMonitor_After_Action	MfgSession	findAssembly_1_internalaction_1
8	EJB_After_SessionEjbBusinessMethods	MfgSession	findAssembly
9	CustomMonitor_After_Action	LargeOrderSession	createLargeOrder_1_externalcallaction_1
10	CustomMonitor_Before_Action	LargeOrderSession	createLargeOrder_1_internalaction_2
11	CustomMonitor_After_Action	LargeOrderSession	createLargeOrder_1_internalaction_2
12	EJB_After_SessionEjbBusinessMethods	LargeOrderSession	createLargeOrder
13	JDBC_Before_Statement_Internal	JDBCConnectionWrapper	prepareStatement
14	JDBC_After_Statement_Internal	JDBCConnectionWrapper	prepareStatement

Table 5.4: Example: Call path event record list for a call to method `createLargeOrder` (see Figure 5.14).

container that decides when a transaction is to be started, suspended, aborted or committed. The mechanism can be configured using transaction attributes. This way, the EJB container is advised when and how an EJB business method should be enlisted in transaction processing. Attributes highly impacting the performance of transactions are the isolation levels of participating resources [**?**]. We do not model the isolation level explicitly, it is implicitly taken into account when measuring the resource demands of a transaction's commit.

5.6.2.2 Quantifying Demands

In Section 3.2, general considerations concerning the extraction of resource demands are presented. There we already explained our two approaches to resource demand quantification. In this section, we explain these approaches in more detail and relate them to the PCM performance model extracted as described in this chapter. The two approaches are: i) use measured response times to estimate CPU resource demand or ii) estimate the CPU resource demand using the Service Demand Law. Given that we aim at an automated extraction during operation, we have to handle a transaction mix.

Response Time

Using measured response times as approximation of the resource demand is simple. Whether the workload consists of one request class or a transaction mix makes no difference for the application of this approach. A severe requirement is, that the resource utilization of the system under consideration has to be low when measuring. Otherwise, waiting times would have a significant impact on the measured response times. For example for CPU resources, if the scheduling mechanism is preemptive, a job under processing might be suspended and put back in the waiting queue. The job is not guaranteed to be processed at once. Coming back to response time measurements, the activities we measure should not be interrupted. The longer the activity is, the higher is the probability that it is not processed in one piece.

For measuring response times we use WLDF. Given that the performance-relevant actions are extracted in separate methods (see Section 5.4.1), monitors observing the internal actions can be configured. Hence, we observe a service's response time on the granularity of internal actions. With reference to the previous section this means that we observe the resource demand at the application server CPU. The resource demand at the DBS CPU can be obtained by measuring the time of the transaction's commit phase. There are two diagnostic actions to measure response times: Action `TraceElapsedTimeAction` generates event records containing the measurement as payload, action `MethodInvocationStatisticsAction` aggregates the measured response time to method-specific statistics that are kept in memory. The latter has the advantage that it has lower overhead than the event record triggering action. The advantage of event records is that each measurement is archived with its context id. The measurements can then be used to derive a realistic distribution function for the response time. Furthermore, the single time measurements can be related to the actual control flow path or related to actual method parameter characterizations. The aggregating action provides only the first two moments of the observed response time distribution. Since there are no context ids available, i.e., the context information of the individual measurements is lost, the obtained time values cannot be related to, e.g., the control flow.

Measurements:

Internal actions:	intAct_1, intAct_2
Observed time interval:	$t = 10\ min = 600\ sec$
Mean CPU utilization:	$\mathcal{U}_{\text{CPU}} = 0.5$ (i.e., 50%)
Invocation count of intAct_1:	$ic(\text{intAct}_1) = 100$
Invocation count of intAct_2:	$ic(\text{intAct}_2) = 200$

Figure 5.15: Example: Measurements for resource demand estimation.

When discussing time measurements, one has to also consider the timer accuracy and resolution. The timer WLDF internally make use of is the Java API method `java.lang.System#nanoTime`. For our purpose, both accuracy and resolution are considered sufficient. For a detailed analysis of the Java API timer we refer the reader to [**?**].

Service Demand Law

In order to apply the service demand law, we need to measure CPU utilization and method count. The application server CPU utilization is partitioned among the processed internal actions, the DBS CPU utilization is partitioned among the transaction commit phases. To determine appropriate fractions, we apportion the CPU utilization using response time ratios or using profiling information. Note that both alternatives entail an increased system overhead, either by measuring response times or profiling the application.

Figure 5.15 shows some exemplary measurements that are necessary for the application of the service demand law. During a time interval of length t, the CPU processes two internal actions whose resource demands are of interest. The mean CPU utilization is $\mathcal{U}_{\text{CPU}}$. The invocations counts of internal actions are given, $ic(x)$ denotes the invocation count of an internal action x during the observed time interval. If we would know the run-time portion $pn(x)$ of an internal action x, we could compute the corresponding resource demand by

$$\frac{\mathcal{U}_{\text{CPU}} \cdot pn(x)}{ic(x)} \ .$$

For a specific resource and a specific time interval, we define an action's run-time portion as the fraction of the total resource utilization that is caused by that action. In order to obtain the run-time portion for each internal action of interest, we investigate different options.

To obtain profiling information we use JRA (see Section 2.2.2). We use method sample counts to determine the run-time portion of each method during a fixed time interval. The sample count of a method states how often the method is sampled during the observation period. We would like JRA to sample a method whenever the method or one of its successors is executed at a sampling point. A method's successor is understood as a method that is synchronously invoked by the considered method itself or by one its successors. To achieve this, JRA's stack trace depth would

Sample counts obtained with JRA:

Sample count of intAct_1: $sc(\text{intAct}_1) = 350$

Sample count of intAct_2: $sc(\text{intAct}_2) = 150$

Total sample count: $total\_sc = 600$

Computing run-time portions with absolute sample counts:

Run-time portion of intAct_1: $pn(\text{intAct}_1) = \frac{sc(\text{intAct}_1)}{total\_sc} = \frac{350}{600} = 0.58$

Run-time portion of intAct_2: $pn(\text{intAct}_2) = \frac{sc(\text{intAct}_2)}{total\_sc} = \frac{150}{600} = 0.25$

Computing run-time portions with relative sample counts:

Run-time portion of intAct_1: $pn(\text{intAct}_1) = \frac{sc(\text{intAct}_1)}{sc(\text{intAct}_1)+sc(\text{intAct}_2)} = \frac{350}{350+150} = 0.7$

Run-time portion of intAct_2: $pn(\text{intAct}_2) = \frac{sc(\text{intAct}_2)}{sc(\text{intAct}_1)+sc(\text{intAct}_2)} = \frac{150}{350+150} = 0.3$

Figure 5.16: Example: Estimating run-time portions based on sample counts.

have to be configured to be greater than or equal to the greatest method stack depth that can occur during profiling. However, even in this case JRA would still miss a method executed at a sampling point if the method is executed in a thread that is not considered as part of the examined threads. This is because JRA does not consider all threads that are active at a sampling point in time but a randomly selected subset. In summary, for a representative JRA recording the sample rate and the stack trace depth have to be high. The downside of such a configuration is, that the sampling overhead increases. Especially because we monitor in the context of EJB applications, the method stack trace is big. It comprises many calls to methods provided by the middleware.

To estimate run-time portions with sampling information, we either use *absolute sample counts* or *relative sample counts*. Figure 5.16 illustrates the two alternatives. There are sample counts for the two internal actions introduced in Figure 5.15, where also the observed time interval is specified. The total sample count does not equal the sum of the internal action sample counts. This means that there are some samples that did not include the methods that represent the internal actions.

Using absolute sample counts, the run-time portion of a method is estimated as the fraction of the total sample count attributed to the method. This alternative assumes that samples not containing the respective internal action's method do not contribute to the internal action's resource demand. This might lead to an underestimation of the internal action's run-time portion. Using relative sample counts, the run-time portion of an internal action is estimated as the ratio of the respective internal action's sample count to the sum of the sample counts of all internal actions of interest. This alternative assumes that samples not containing a method of any internal action still do contribute to the resource demand of the considered internal actions. Those samples are apportioned among all considered internal actions. Thus, in contrast to the absolute sample count alternative, this option requires profiling of all running internal actions.

Average response times:

Response time of intAct_1: $\mathcal{R}(\text{intAct}_1) = 1.8\ sec$

Response time of intAct_2: $\mathcal{R}(\text{intAct}_2) = 0.4\ sec$

Computing run-time portions with weighted response time ratios:

$$\text{Run-time pn. of intAct}_1: \quad pn(\text{intAct}_1) = \frac{ic(\text{intAct}_1)\cdot\mathcal{R}(\text{intAct}_1)}{ic(\text{intAct}_1)\cdot\mathcal{R}(\text{intAct}_1)+ic(\text{intAct}_2)\cdot\mathcal{R}(\text{intAct}_2)} = \frac{100\cdot 1.8}{100\cdot 1.8+200\cdot 0.4} = 0.69$$

$$\text{Run-time pn. of intAct}_2: \quad pn(\text{intAct}_2) = \frac{ic(\text{intAct}_2)\cdot\mathcal{R}(\text{intAct}_2)}{ic(\text{intAct}_1)\cdot\mathcal{R}(\text{intAct}_1)+ic(\text{intAct}_2)\cdot\mathcal{R}(\text{intAct}_2)} = \frac{200\cdot 0.4}{100\cdot 1.8+200\cdot 0.4} = 0.31$$

Figure 5.17: Example: Estimating run-time portions based on response times.

Another option to estimate run-time portions is illustrated in Figure 5.17. Here, we use the internal action's response time weighted by their invocation counts to determine the run-time portion. Given that we monitor during a fixed time interval, the weighting by invocation counts can also be understood as weighting by throughputs. Similar to the relative sample count alternative, this option requires instrumenting of all running internal actions. In the example, it is assumed that only the two internal actions are causing load. Partitioning the workload based on the response times implicitly assumes that a method with an x times greater response time than another method also has an x times greater resource demand. Since we claim an internal action to only use the application server CPU, this assumption is considered appropriate.

5.7 Obtaining Workload and Resource Utilization Data

Given that the observation takes place during run-time, monitoring workload data is possible. WLDF provides access to system load information as well as data about resource utilization. We distinguish between workload at the system level and workload at the individual component level. For both levels we also distinguish between workload in terms of user requests and workload in terms of resource consumption.

At the system level for instance, the utilization of physical memory, Java heap memory, JDBC connection pools, WLS thread pools and CPUs can be monitored. There are MBeans like the `JRockitRuntimeMBean` providing information about the run-time state of the system. Such MBeans can be accessed using the data harvester or directly through JMX. User requests at the system boundary can be tracked to compute service request arrival rates and provide information about the service input parameters. Therefore, the WLDF instrumentation engine has to be configured to instrument at the system boundaries and monitor the input parameters using the diagnostic action `DisplayArgumentsAction`.

The above also applies to individual components. Considering the component boundaries, request tracking to compute component-specific request arrival rates and mon-

itor the input parameters is possible. Furthermore, at the component level, the EJB instance pool utilization can be monitored. This is done by accessing the Runtime MBean `EJBPoolRuntimeMBean` representing the pool of a stateless session bean or a message-driven bean, or by accessing `EJBCacheRuntimeMBean` representing the cache of a stateful session bean.

The workload and resource utilization data does not serve as input data for our performance model. Besides the PCM usage model, PCM does not explicitly allow incorporating workload information. In PCM, workload information is made available by means of simulation. Obtaining workload information is not our primary focus but nevertheless, such information can be used to detect performance hot spots or bottlenecks.

5.8 Limitations

In this section, we describe the limitations of the presented approach to model extraction.

We extract components, inter-component control flow and intra-component control flow using call path tracing. In order to obtain representative call paths, a representative workload has to be available. Only those parts of the application that are actually running during observation are considered in the extraction. However, given that monitoring is intended to be performed during operation, the proposed approach will result in extracting the "effective architecture". Parts of the application that are not called during operation are also not modelled.

For the extraction, we assume performance-relevant information about component boundaries and intra-component control flow to be available. Note that this information could be provided automatically by a static analysis. Furthermore, due to the lack of tools supporting in-method instrumentation, we extract performance-relevant fragments in separate methods as a workaround. The code refactoring can be automated. For instance, a chaining of ArchiRec [**?**] and an adapted Java2PCM [**?**] could be employed not only to detect performance-relevant information but also to refactor according to the schema proposed in Section 5.4. See Chapter 4 for more information on ArchiRec and Java2PCM. Additionally, our approach does not allow for the extraction of explicit parametric dependencies. Instead, dependencies between input parameter characterizations and the control flow can be observed. The consideration of parametric dependencies is limited to statistical reports describing the relation between observed parameter tuples and the monitored control flow. However, since we use PCM as performance model, manual adaptations are easily possible.

The resource demands we extract from measurements conducted in a specific system environment. Thus, the resource demands are environment-specific. On the other hand, given that the estimated resource demands stem from actual measurements, we expect the performance predictions to be realistic. The system environment is currently limited to one WLS instance. However, this limitation can be relaxed if event records from multiple servers can be chronologically ordered.

The PCM resource environment model as well as the PCM allocation model have to be provided manually by the user. However, these models have a much lower complexity than the PCM repository model and PCM system model which we extract. Furthermore, we do not model memory consumption. Instead, monitoring

data about resource utilization can be obtained. However, even though we monitor only the WLS instance we estimate the utilization of both the application server CPU and the DBS CPU.

6. Tool Prototype

This chapter presents how the extraction method described in Chapter 5 is implemented as a tool prototype. Starting with an overview on the prototype's architecture, we describe the implemented extraction steps. A section on limitations concludes this chapter.

6.1 Architecture

The architecture of the extraction tool is illustrated in Figure 6.1. The Java EE application of interest is deployed on a WLS instance that is running in a JRockit JVM. The application, which may access a database, is monitored by WLDF, monitoring data is sent to the diagnostic data store. The application's workload is generated outside WLS. The tool prototype accesses both WLS run-time information and diagnostic data through a Runtime MBean server and may additionally access profiling data from the JRockit Runtime Analyzer (JRA). The prototype then processes the information and extracts a PCM 3.0 instance.

The illustration shows that the diagnostic data store is running inside the WLS instance, i.e., it is configured as file-based store. We prefer the file-based store over the JDBC-based store because it typically has lower overhead.

6.2 Extraction Steps

The extraction during operation is organized in several steps. "At a high level", a fully-functioning extraction tool implementing the method of Chapter 5 should operate according to the following steps:

- A WLS instance in which the application of interest is deployed is assumed to be running.
- The system developer specifies which parts of the application he wants to extract models for.

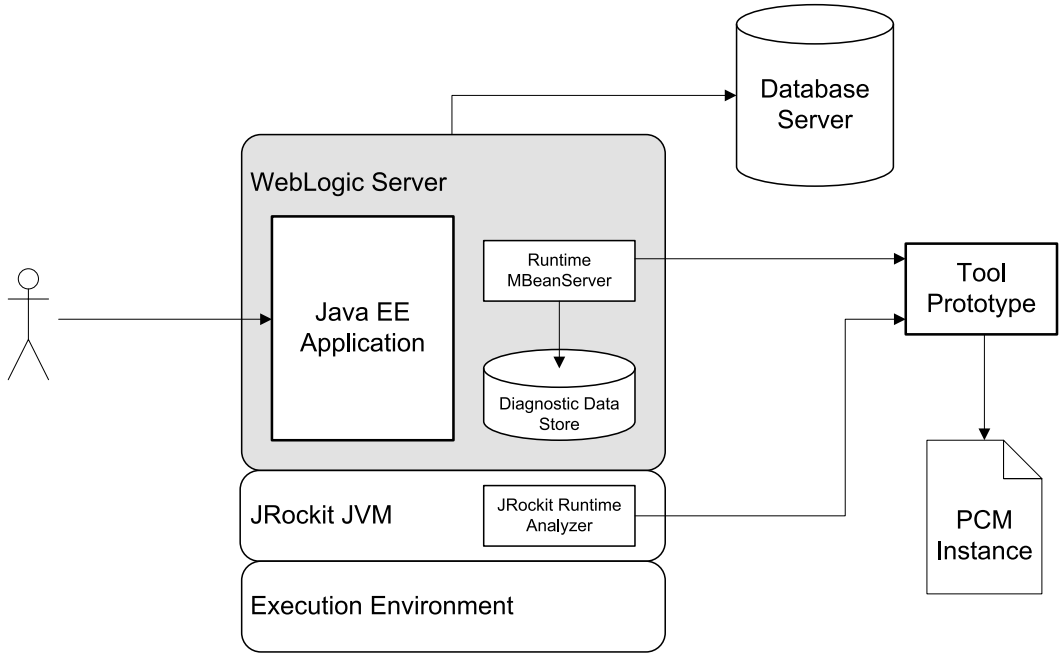


Figure 6.1: Architecture of the extraction tool.

- The tool automatically configures monitoring and instrumentation tools, e.g., WLDF.
- The application is monitored during operation.
- Based on the monitoring data, the tool extracts a PCM instance.

Obviously, this is only a rough concept. The following two sections present the steps in more detail. The model extraction process is related to structural extraction respectively resource demand extraction. The PCM generation is conducted semi-automatically, it will require some intervention by the user (the system developer).

6.2.1 Extracting Structure

In this section we describe the steps in which the tool prototype extracts the PCM performance model. The output is a PCM repository model and a PCM system model. The extraction of the resource demands is discussed in the next section. Building a PCM resource environment model, a PCM allocation model and a PCM usage model is not considered here.

0. We assume the application of interest to be refactored according to the schema proposed in Section 5.4 and running on a single WLS instance.
1. At first, the prototype's user has to specify to which WLS instance to connect. Since we connect via JMX, a location consisting of a URL and a port, a protocol and user credentials have to be configured.
2. The extraction then begins with a specification of the component boundaries by the user as described in Section 5.2. The specification has to be provided as

```
<?xml version="1.0" encoding="UTF-8"?>

<!-- This is a sample system description a user has to provide.
     It describes the SPECjAppServer2004_Next benchmark application. -->

<fb:system xmlns:fb="http://sdq.ipd.uni-karlsruhe.de/fabro"
           xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">

  <!-- Choose application with name specj. -->
  <fb:application>specj</fb:application>

  <!-- Choose to map each EJB to its own component. -->
  <fb:configmode>ejbtocomponent</fb:configmode>

</fb:system>
```

Figure 6.2: Example: System description provided as XML document.

XML document conforming to an XML schema. We show an exemplary XML document in Figure 6.2. In terms of the tool prototype, such an XML document is denoted as "system description". The example describes the SPECjAppServer2004_Next benchmark application. The user provides the name under which the application is deployed on the WLS instance and selects how the EJBs are grouped to components. In the given example, each component consists of exactly one EJB. The XML schema allows to choose between the three modes `ejbtocomponent`, `packagetocomponent` and `manual`. In the latter case, the components can be each specified by a name and a set of containing EJBs. The corresponding XML schema is presented in Appendix A.1.

3. In the next step, a connection to the WLS instance is established. For the application with the given application name the prototype reads all EJBs and groups them to components as previously specified by the user. Furthermore, the fully-qualified names of the classes implementing the EJBs have to be obtained.

4. Thereafter, according to Section 5.3 and Section 5.4, WLDF has to be configured both for i) the inter-component control flow extraction and ii) the intra-component control flow extraction.

   i) The EJB business methods have to be instrumented. Thus, the application-scope delegating monitors `EJB_Before_SessionEjbBusinessMethods` and `EJB_Before_SessionEjbBusinessMethods` with diagnostic action `DisplayArgumentsAction` have to be configured. Only those classes that have been identified in the previous step as EJB implementation classes have to be instrumented.

   ii) Those methods, that have been extracted according to the naming schema of Section 5.4, have to be instrumented. Therefore, we instrument all methods contained in a class that implements an EJB. This is done by two custom monitors with action `TraceActon`. One monitor captures the method entries and one monitor captures the method exits. Figure 6.3 shows an XML fragment defining the monitor `CustomMonitor_IntraCCF_Before` to instrument the entries of those methods that are contained in, e.g., class `WorkOrderSession` or class `OrderSession`.

```
<!--## intra-component control flow extraction monitor ##-->
<wldf-instrumentation-monitor>
  <name>CustomMonitor_IntraCCF_Before</name>
  <enabled>true</enabled>
  <action>TraceAction</action>
  <location-type>before</location-type>
  <pointcut>
    execution(* org.spec.jappserver.ejb.mfg.session.WorkOrderSession *(...))
    OR
    execution(* org.spec.jappserver.ejb.mfg.session.MessageSenderSession *(...))
    <!-- ... -->
    OR
    execution(* org.spec.jappserver.ejb.orders.session.OrderSession *(...))
  </pointcut>
</wldf-instrumentation-monitor>
```

Figure 6.3: Example: Custom monitor for intra-component control flow extraction.

If the diagnostic monitors are already woven into the application code but disabled, the monitors only have to be enabled. If the monitors have to be newly woven into the code, the application has to be redeployed. This is because we assume the hot swap feature to be disabled (see Section 2.2.1.2).

5. The application of interest is now monitored. The application may be monitored during operation or workload may be injected manually. The workload should be representative in terms of the kind of system requests. The workload is not required to be representative in terms of request arrival rates.

6. The prototype's actual extraction method has to be triggered now. It extracts a PCM repository model and a PCM system model as described in Chapter 5. The WLDF archive is accessed via the WLDF data access Runtime MBean. A still pending issue is the question of how to map data types to PCM data types. We implemented the following procedure: Java primitive types as well as their Java wrapper classes are mapped to the PCM primitive types. This applies also for the Java String type. Java array types are mapped to a PCM collection type with an element type according to the array types's element type. Java classes or interfaces inheriting the `java.util.Collection` interface are also mapped to a PCM collection type. Here, the element type always refers to the PCM type representing `java.lang.Object`. This is because the event record field describing the method parameters (see Table 2.1) does not provide information about the element type of the generic collection interface. Not only `java.lang.Object` but all other types that are not covered yet are mapped to a PCM composite type without any inner types.

6.2.2 Extracting Resource Demands

In this section, the necessary steps to extract the resource demands are presented. We assume the preceding steps of the previous section to be already processed. Thus, the steps are numbered consecutively.

7. Here, WLDF has to be configured to extract resource demands (see Section 5.6). Therefore, we configure WLDF to inject a time sensor at the internal actions. This is done by a custom monitor we define according to the example

```
<!--## resource demand extraction monitor ##-->
<wldf-instrumentation-monitor>
  <name>CustomMonitor_MeasureResourceDemand_Around</name>
  <enabled>true</enabled>
  <action>MethodInvocationStatisticsAction</action>
  <location-type>around</location-type>
  <pointcut>
    execution(* org.spec.jappserver.ejb.mfg.session
                .WorkOrderSession *_internalaction_*(...))
    OR
    execution(* org.spec.jappserver.ejb.mfg.session
                .MessageSenderSession *_internalaction_*(...))
    <!-- ... -->
    OR
    execution(* org.spec.jappserver.ejb.orders.session
                .OrderSession *_internalaction_*(...))
  </pointcut>
</wldf-instrumentation-monitor>
```

Figure 6.4: Example: Custom monitor for resource demand extraction.

given in Figure 6.4. The XML fragment defines a monitor with location type `around` and diagnostic action `MethodInvocationStatisticsAction`. This action does not generate event records. Instead, to minimize the monitoring overhead, statistics about the response times are aggregated at run-time. We evaluate the monitoring overhead in Section 7.3.1. In the example, the internal actions of, e.g., class `WorkOrderSession` are measured. More precisely, we measure the internal actions of the component comprising an EJB that is implemented by class `WorkOrderSession`.

The monitors introduced in step 4 now are to be disabled. When monitoring resource demands, the monitoring overhead should be as small as possible. To avoid a redeployment of the application, we propose the following: The resource demand extraction monitor should be configured within step 4, but first as a disabled monitor. When this step, i.e., step 7 is reached, that monitor is enabled while the other event record generating monitors are disabled. This way, the reconfiguration can be carried out without an interruption of the application.

8. The application of interest is now monitored. The application may be monitored during operation or workload may be injected manually.

9. Finally, the resource demand is estimated. We implemented two approaches that are presented in Section 5.6.2.2: i) We use response times to approximate resource demands or ii) we use the Service Demand Law and partition with weighted response time ratios.

   i) The measured average response times are accessed via the WLDF instrumentation Runtime MBean. Then the method-specific response times are attached to those internal actions the methods represent. Note that this approach is only applicable for low system load.

   ii) In order to apply the Service Demand Law, we need the length of the observed time interval and the utilization of the application server CPU. The observed time interval is the monitoring time of step 8. If the data is provided, we partition the resource utilization by weighted response

time ratios. The method-specific response times and invocation counts are accessed via the WLDF instrumentation Runtime MBean. Note that this approach is only applicable for medium to heavy system load.

This way, we only extract resource demands for the application server CPU. In Section 5.6.2 we also described how to derive resource demands for the DBS CPU by measuring commit phases of transactions. Time measurements of commit phases cannot be conducted directly because in general, transaction boundaries are not externalized. Bean-managed transactions have to be measured differently than container-managed transactions which themselves have to be differentiated by combinations of transaction attributes.

Thus, solutions to capture resource demands on the DBS have to be tailored for each application. For bean-managed transactions, the delegating, application-scope monitor `JTA_Around_Commit` can be used. With a time-measuring diagnostic action, the time of a transaction's commit phase then can be measured. Container-managed transactions necessitate a different proceeding. In the following, we describe a common case. Assuming there is a session bean with a business method `meth` that enforces the EJB container to always start a new transaction whenever `meth` is processed. Hence, the transaction always ends after the method exits. In order to measure the transaction's commit phase, a method `meth′` delegating to `meth` is introduced. All calls to `meth` have to be redirected to `meth′`. The method `meth′` has to be annotated with transaction attribute `NOT_SUPPORTED`. The transaction's commit phase can then be measured as the difference between the response times of `meth′` and `meth`. Note that a detection of a rollback instead of a commit is not discussed here.

We did not implement the JRA alternatives to quantify resource demands as they are presented in Section 5.6.2.2. Nevertheless, we evaluate the alternatives in Section 7.3.3.

6.3 Implementation

The prototype developed in the context of this diploma thesis is intended as a proof-of-concept implementation and focuses on the main issues. The tool prototype does not provide a GUI and it does not support automated configuration of WLDF. Referring to the steps presented in the previous section, the steps 3, 6 and 9 are automated. In fact, the actually extracting steps 6 and 9 are processed both at the end directly one after another. In the previous section they are separated only for reasons of clarity.

In this section, we provide some information on the prototype's implementation. It is not intended to provide details but to give an overview.

The tool is implemented as a Eclipse plug-in in Java. In order to get an overview on how the implementation is organized, Figure 6.5 shows the package structure. Only the main packages and their dependencies are shown. The following list provides a short documentation for each package. Subpackages are not considered here.

- `de.uka.ipd.sdq.fabro.pcm` This package contains adapters to the PCM repository factories and PCM system factories. These adapters are used to create PCM instances.

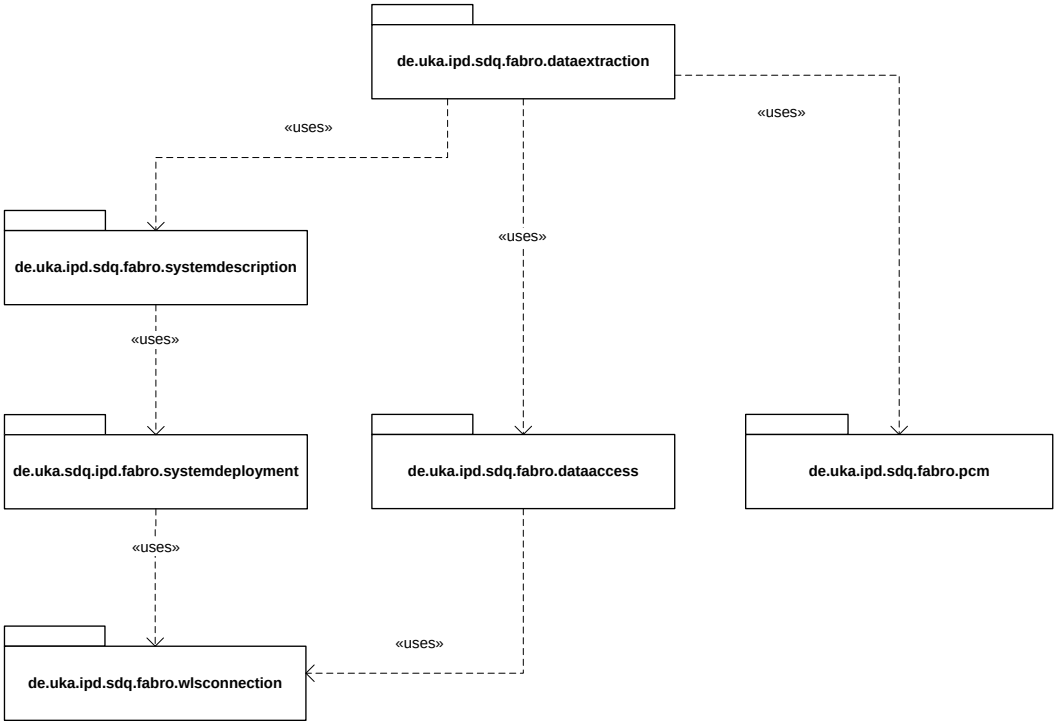


Figure 6.5: Package overview of the tool prototype.

- `de.uka.ipd.sdq.fabro.wlsconnection` This package provides functionality to establish connections to WLS instances. The connections are JMX connections to the Domain Runtime MBean server.
- `de.uka.ipd.sdq.fabro.dataaccess` With this package, monitoring data, i.e., WLDF diagnostics data can be accessed. That requires WLS connections.
- `de.uka.ipd.sdq.fabro.systemdeployment` This package provides read-only access to deployment information about Java enterprise applications and their EJBs running on a WLS instance. For instance, with the help of this package, questions like "Which EJBs are currently deployed?" can be answered. Note that the term "systemdeployment" here refers to the deployment configuration of a WLS instance. It does not deal with the PCM system model.
- `de.uka.ipd.sdq.fabro.systemdescription` Descriptive, this package acts as a mediator between the system specification as it is provided by the user and the current deployments on the considered WLS instance. Classes implemented in this package read the system specification, adjust the specification with the current deployment information and finally provide a data structure that describes the system to analyze.
- `de.uka.ipd.sdq.fabro.dataextraction` The logic of the extraction is contained in this package. It accesses monitoring data and extracts a PCM instance based on the system description.

Information on the internal structure of the mentioned packages can be found in the Appendix A.2.

6.4 Limitations

In this section, we outline limitations that concern the prototype implementation as presented in this chapter.

The implemented prototype does not provide a GUI that simplifies its usage. Furthermore, an automated configuration of WLDF is not implemented. Nevertheless, even if it was implemented, the drawback of a required redeployment remains. Since we intend to monitor a Java EE application during operation, redeploying an application is an issue. However, the application has to be redeployed only once. The whole extraction method can then be processed repeatedly. Only if there are structural changes to the application, the monitors have to be adapted. In that case, a redeployment would be necessary anyway.

In addition, an automated handling of message-driven beans is not yet implemented. Demands for the application server CPU are extracted while demands for the DBS CPU are not automatically captured. This is due to the challenge of determining transaction boundaries and making response times of commit phases measurable.

The extraction of parametric dependencies as proposed in Section 5.4.2 is not implemented. The same applies for the workload monitoring in Section 5.7 and the automated extraction of passive resources.

7. Evaluation

In this chapter we evaluate the method we proposed in Chapter 5. The evaluation is based on the SPECjAppServer2004_Next benchmark, a beta version of the successor of SPECjAppServer2004. While Section 2.5 gives a general overview on the setting underlying the benchmark, the subsequent Section 7.1 focuses on the technical implementation. Section 7.2 describes the system environment in which we deployed the benchmark. Section 7.3 shows several experiments to answer questions regarding the overhead of monitoring. In Section 7.4 we conduct a case study with the tool prototype to evaluate the results our extraction method delivers. A section on the evaluation's findings concludes this chapter.

7.1 SPECjAppServer2004_Next

We divide the presentation of implementation details of the considered SPECjAppServer2004_Next benchmark into a description of the benchmark application and a description of the benchmark driver. For the benchmark's architecture see Section 2.5.

7.1.1 Benchmark Application

The benchmark application is implemented as a Java EE application. It is claimed that the application has similar requirements on the execution environment as a typical, real-world enterprise application. There are three domains that are deployed on the application server under test: the orders domain, the manufacturing domain and the supplier domain. For the communication between the domains JMS queues are used. The benchmark driver accesses only two domains, namely the orders domain and the manufacturing domain. The former is accessed via Java Servlets, the latter either by RMI or by web service calls.

In this evaluation we focus on the manufacturing domain because our tool prototype does not support Servlets and JMS messaging yet. Figure 7.1 shows the four stateless session beans and two message-driven beans the manufacturing domain consists of. Together with the `LargeOrderSession` EJB, the MDB `LargerOrderMDB` processes orders sent by the orders domain. The MDB `ReceiveMDB` processes delivery

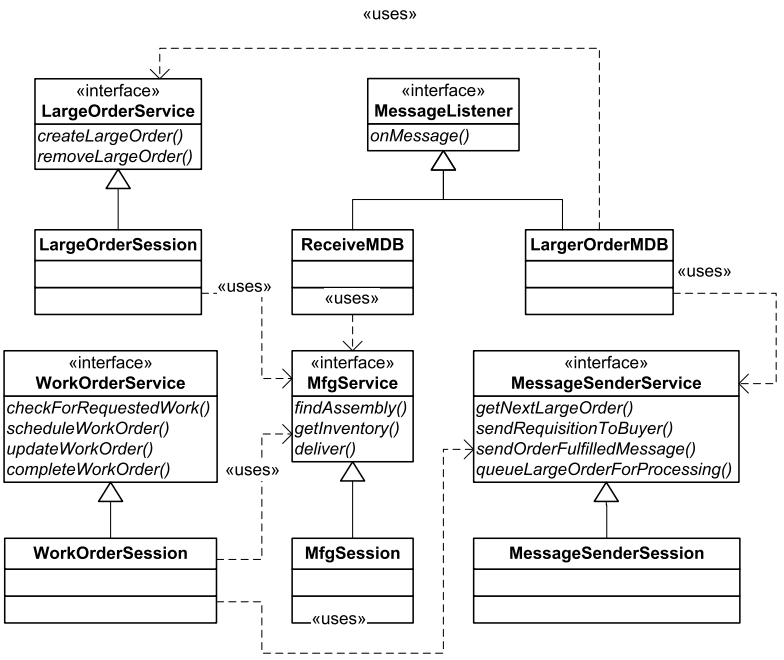


Figure 7.1: UML class diagram of the benchmark's manufacturing domain.

information from the supplier domain. The EJB `WorkOrderSession` handles work order processing and is the manufacturing domain's main EJB. It is the only bean of the manufacturing domain that is called by the benchmark driver. The beans `MfgSession` and `MessageSenderSession` act as helper beans.

7.1.2 Benchmark Driver

The SPECjAppServer2004_Next benchmark driver implementation is based on the Faban "facility for developing and running benchmarks" [**?**]. The Faban driver framework simplifies benchmark development, supports stochastic models for user simulations and controls the life cycle of a benchmark run.

In Faban, a benchmark may consist of several driver classes implementing several benchmark operations. For the benchmark one can configure a ramp up time, a steady state time and a ramp down time. The steady state time describes the interval in which the system under test is actually observed. For a benchmark driver one can configure the number of executing threads, customize a driver report and specify the benchmark operations to be processed. Faban allows to regulate the benchmark driver's cycle time by means of probability distributions. For instance, with the exponential distribution it is possible to simulate a Poisson process. The benchmark operation executed at the begin of a cycle is determined by an operation mix. Benchmark operations are implemented as simple Java methods.

The Faban driver framework provides timer functionality to allow specifying how elapsed times of benchmark operations are to be measured. One way is the manual definition of a critical section of the method implementing the benchmark operation. In terms of Faban, a critical section is the section whose response time should be gathered, i.e., the section that contains those actions that are actually benchmarked.

The benchmark we consider for evaluation purposes provides two benchmark drivers: one driver simulating manufacturing sites and one driver simulating dealerships.

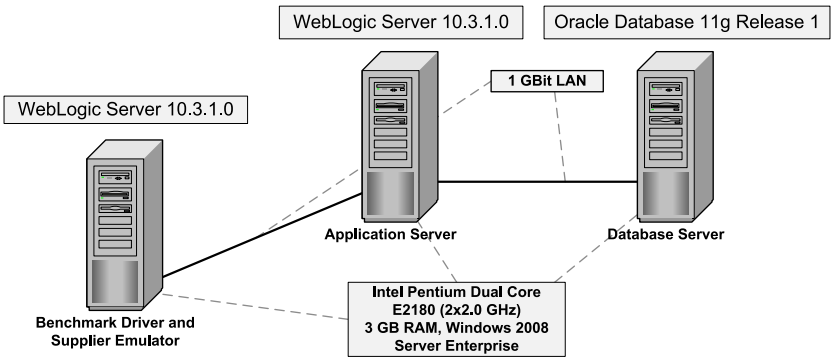


Figure 7.2: System environment.

Both drivers simulate the requests by means of a Poisson process. The benchmark report aggregates several measurements to a single metric, namely business operations per seconds (BOPS). This metric can then be used to compare the performance of Java application servers. To ensure comparability there are rules on how the system under test has to be set up and configured before running the benchmark. To put it simply, the whole benchmark including the driver and the application can be configured by a specific injection rate. The injection rate determines the amount of data in the database, the configuration of the application server and the workload the driver generates. It is a scaling factor to enable the deployment in execution environments of different dimensions.

In the context of the evaluation, we adapted the mentioned benchmark drivers respectively implemented own drivers. For instance, we disabled the driver that produces workload for the dealer domain. During evaluation, we found an error concerning the response time measurements conducted by Faban. The times measured with millisecond granularity were incorrect. On current Microsoft Windows operating systems the timer accessed by the Java timer method `System#currentTimeMillis` has an accuracy of about 15 ms. This is shown in [**?**]. To calibrate its nanosecond timer Faban made use of the millisecond timer assuming a 1 ms accuracy. Thus, the response times measured with the nanosecond timer deviated by several milliseconds. In a new version of the Faban driver framework the issue was corrected.

7.2 Experimental Environment and Experiment Setup

For the evaluation, the benchmark is deployed in the system environment illustrated in Figure 7.2. There are three machines: The Java EE benchmark application is deployed on a WebLogic Server (WLS) instance of version 10.3.1.0. As a database server (DBS), we use Oracle Database 11g Release 1. The system under test consists of both the WLS instance and the DBS. The benchmark driver as well as the supplier emulator run on a separate machine. The machines are connected via 1 GBit Ethernet. As operating system, Microsoft Windows Server 2008 Enterprise is used.

Note that the CPUs are dual-core CPUs. Since the PCM Bench 3.0 respectively the simulation framework SimuCom currently does not support multi-core CPUs, we had to switch one core off. In terms of queueing networks a multi-core CPU

has to be mapped to a multiserver queue. A multiserver queue provides one queue for multiple servers. Currently, SimuCom is capable of simulating multiple CPUs, but only with exactly one queue for each CPU. Thus, multi-core CPUs cannot be simulated.

Furthermore, the CPUs we use support the Enhanced Intel Speedstep Technology (EIST). EIST aims at decreasing power consumption and average heat production by means of dynamic frequency and voltage scaling. In short, the processor speed varies according to the processor utilization. The less the CPU is utilized the slower it works. With EIST, from the user point of view, an application seems to have load-dependent resource demands. Obviously, the resource demands remain the same, it is the resource's processing rate that changes. We have to switch EIST off because we do not model dynamic frequency scaling. Currently, the PCM Bench does not provide such a feature.

As mentioned in Section 2.3, we use the perfmon tool to measure resource utilizations. When conducting an experiment we specify a ramp up time after which we assume a steady system state to be reached. We measure the utilizations (values between 0.0 and 1.0) only during steady state time. Response times or method invocation counts are also measured during steady state time. Both elapsed times and invocation counts are either measured by the driver at the client side or with WLDF with action `MethodInvocationStatisticsAction`. We introduce the following notation:

$\mathcal{U}_{\text{DBS\_CPU}}$:	Average utilization of the DBS machine's CPU.
$\mathcal{U}_{\text{WLS\_CPU}}$:	Average utilization of the WLS machine's CPU.
$\mathcal{U}_{\text{DBS\_IO}}$:	Average utilization of the DBS machine's disk.
$\mathcal{U}_{\text{CLIENT\_CPU}}$:	Average utilization of the client machine's CPU.
$\mathcal{D}_{\text{WLS\_CPU}}$:	Demand for the WLS machine's CPU.
$\mathcal{D}_{\text{DBS\_CPU}}$:	Demand for the DBS machine's CPU.
$\mathcal{R}_{\text{Driver}}$:	Average response time measured by the benchmark driver.
$\mathcal{R}_{\text{WLDF}}$:	Average response time measured by WLDF.
ic_{Driver} :	Invocation count monitored by the benchmark driver.
ic_{WLDF} :	Invocation count monitored by WLDF.

Furthermore, to ensure repeatability, for each experiment the WLS instance is set up from scratch. For exemplary experiments see the subsequent section where different monitoring approaches are assessed.

7.3 Evaluating Monitoring Approaches

In this section, we conduct experiments to assess the overhead of the WLDF instrumentation, to figure out if and how JVM optimizations influence the instrumentation and to assess the method sampling approaches for resource demand quantification. The experiments are used to motivate the design decisions we made in Chapter 6.

7.3.1 Instrumentation Overhead

To evaluate the overhead of the WLDF instrumentation, we run the benchmark application with different diagnostic actions injected. By comparing resource utilization and the benchmark operation's response times we get an overview to what extent instrumentation affects an application's performance.

SPECjAppServer2004_Next benchmark (txRate = 5, ramp up time = 600 sec, steady state time = 1140 sec, ramp down time = 60 sec).									
All EJB business methods instrumented with action:	$\mathcal{U}_{\mathbf{WLS}}$	$\mathcal{U}_{\mathbf{DBS}}$		$\mathcal{U}_{\mathbf{CLIENT}}$	$\mathcal{R}_{\mathbf{Driver}}$ [ms]				
	CPU	CPU	IO	CPU	Purchase	Manage	Browse	Create-VehicleEJB	Create-VehicleWS
-none-	0.519	0.163	0.004	0.029	62	74	403	1108	1118
`MethodInvocationStatistics-Action`	0.549	0.163	0.006	0.025	74	77	426	1131	1138
`TraceElapsedTime-Action`	0.607	0.164	0.003	0.023	432	361	964	1518	1559

Table 7.1: Experiments regarding WLDF instrumentation overhead.

We investigated three different instrumentation configurations: i) The application is not instrumented at all, ii) all EJB business methods are monitored with action `MethodInvocationStatisticsAction` or iii) all EJB business methods are monitored with action `TraceElapsedTimeAction`. Both considered diagnostic actions measure response times. The latter creates an event record for each measurement whereas the former aggregates the measurements in statistics that are kept in memory.

Table 7.1 shows the experimental results. The Configurations ii) and iii) both introduce an overhead. The utilization of the DBS and the client remain almost constant. The utilization of the application server CPU increased from about 50% to 60% leading to a significant increase in the benchmark operation response times. Particularly the event record generating action entails a highly increased response time. Note that the response times for the operations `CreateVehicleEJB` and `CreateVehicleWS` both include an internal delay of about 1000 ms. Hence, the increase from 1108 ms to 1518 ms represents an increase by 480% and not 37%, if we consider the times spent waiting for a response from the server.

We conclude that event record generating actions have a significant impact on the instrumented application. Data aggregating monitors have an overhead as well but it is by far much lower. While we consider the latter overhead acceptable during operation, event record generating actions should only be injected in time frames where the application's response time is not crucial. We do not quantify the instrumentation overhead because it obviously depends on the number of instrumented methods and the user behavior. Compared to the instrumentation required for the extraction as described in Section 6.2, instrumenting only the EJB business methods is a light instrumentation.

As a secondary aspect we conclude that the CPU utilization of the client machine as well as the IO utilization of the DBS are negligible in our experiment setup.

7.3.2 Instrumentation versus JRockit Optimizations

In our experimental setup, the WLS instance runs on a JRockit JVM. One specific characteristic of the JRockit JVM is that there is no interpreter. JRockit runs Just-In-Time (JIT) compilation when a method is called for the first time and then "calls that method's compiled code instead of trying to interpret it" [**?**]. JRockit internally uses method sampling to detect heavily used methods, so-called *hot* methods. Hot methods are then marked for further code optimization like, e.g., method inlining or loop unrolling.

Especially in view of the methods we introduce when refactoring component implementations we have to figure out how JRockit optimizations affect diagnostic actions we inject into the application code. Due to the lack of knowledge concerning JRockit internals we conduct an experimental analysis. To find out if an injected action can be optimized away by JRockit, we first instrument different methods with different actions. Then we run a specific driver that logs method call counts. Comparing the method call counts obtained through instrumentation and the method call counts logged by the driver we can deduce if an action was optimized away. See Section B.1 in the appendix for more detailed information on the experiments we conducted.

We observe that the action `MethodInvocationStatisticsAction` is not affected by JRockit optimizations while action `TraceElapsedTimeAction` is. Obviously, this behavior cannot be guaranteed to hold in all cases, in particular since we can not explain why the results of the two actions differ. Nevertheless, in all experiments we made, action `MethodInvocationStatisticsAction` was never optimized away. The action appears to be immune against the JRockit optimizations. As a conclusion, we propose to disable JRockit optimizations when extracting the application's structure as described in Section 6.2.1. At least the instrumented methods should be excluded from the optimization (JRockit supports such a fine-grained configuration). For the resource demand extraction the optimizations should be enabled. In that step, we only instrument with diagnostic action `MethodInvocationStatisticsAction`.

7.3.3 Method Sampling for Workload Partitioning

In Section 5.6.2.2 different methods for resource demand estimation are proposed. In this section, we conduct experiments to assess the approach of using JRA for run-time portion estimation. We define the run-time portion of a method as the fraction of the total resource utilization that is caused by the method. Run-time portions are needed to partition the resource utilization among different methods. Partitioning is needed to apply the Service Demand Law.

During the experiments we execute a transaction mix of the two business methods `scheduleWorkOrder` (sWO) and `createLargeOrder` (cLO). Both methods trigger internal actions whose resource demands are to be estimated. To identify the internal actions, we introduce the following abbreviations:

sWO_int_2:	`WorkOrderSession#scheduleWorkOrder_1_internalaction_2`
sWO_int_6:	`WorkOrderSession#scheduleWorkOrder_1_internalaction_6`
getInv_int_1:	`MfgSession#getInventory_1_internalaction_1`
fA_int_1:	`MfgSession#findAssembly_1_internalaction_1`
cLO_int_2:	`LargeOrderSession#createLargeOrder_1_internalaction_2`

Transaction mix (ramp up time = 600 sec, steady state time = 1140 sec, ramp down time = 60 sec). During experiments 3-5 JRA recorded 18 min within steady state time with a trace depth of 128 and a sampling interval of 16 ms.									
ID	**Mix**	**JRA**	$\mathcal{U}_{\mathbf{WLS}}$	$\mathcal{U}_{\mathbf{DBS}}$		$ic_{\mathbf{Driver}}$		$\mathcal{R}_{\mathbf{Driver}}$ [ms]	
	sWO : cLO	on/off	CPU	CPU	IO	sWO	cLO	sWO	cLO
1	1 : 2	off	0.088	0.055	0.005	5852	11356	21.41	5.54
2	1 : 2	off	0.454	0.197	0.018	28402	57282	32.48	10.25
3	1 : 2	on	0.495	0.186	0.018	28721	57047	33.42	10.39
4	1 : 1	on	0.524	0.171	0.016	33901	34469	35.25	12.05
5	1 : 4	on	0.539	0.213	0.030	27108	108829	37.84	12.36

Table 7.2: Experiments for resource demand estimation.

Note that getInv_int_1 is only called by sWO whereas fA_int_1 is called by both sWO and cLO. We adapted the implementation of sWO to never trigger the supplier domain, i.e., when calling sWO and cLO only the two methods cause system load.

Table 7.2 describes five experiments we conducted. The experiment with ID 1 is executed under light load. The measured internal actions's response times should be roughly equal to the estimated resource demands [**?**]. Experiments 2 to 5 are executed under medium load (≈50%). Experiments 2 and 3 differ in the usage of JRA, Experiments 3 to 5 differ in the transaction mix. Comparing Experiments 2 and 3 we see that the usage of JRA with a trace depth of 128 and a sampling interval of 16 ms has an acceptable overhead on the WLS CPU. Note that method inlining by the JRockit optimizer has to be avoided during method sampling.

In the following, we compare the JRA alternatives for resource utilization partitioning: i) partitioning with relative sample counts or ii) partitioning with absolute sample counts. In the description of alternative ii) in Section 5.6.2.2 an internal action's sample count is divided by the total sample count. Here, we divide an internal action's sample count by the sample count of method `run` of class `weblogic.work.ExecuteThread`. In a WLS instance, all threads doing work in a Java EE application stem from that method.

Table 7.3 details the resource demand estimations for Experiment 3. For instance, alternative ii) estimates the WLS CPU demand of internal action fA_int_1 to $564300 \cdot \frac{504}{3565} \cdot \frac{1}{85768} = 0.93$. As another example, alternative i) estimates the WLS CPU demand of internal action getInv_int_1 to $564300 \cdot \frac{919}{1968} \cdot \frac{1}{286422} = 0.92$. Notice that the JRA total sample count is 4468 while the `weblogic.work.ExecuteThread#run` sample count is 3565. On the one hand, we cannot explain the notable difference between the total sample count and the worker thread method count. On the other hand, the sample count is much lower than we expected. We assumed JRA to sample each 16+ms a selection of active threads. Even if we consider that $\mathcal{U}_{\text{WLS\_CPU}}$ is about 50% we assumed a total sample count of at least $\frac{1000ms/sec}{16ms} \cdot 0.50 \cdot 1140sec \approx 35000$. It might be that JRA adjusts the sampling rate so that the overhead is kept small. Some further investigations are required.

Table 7.4 provides an overview on the results of alternatives i) and ii) for the Experiments 3 to 5. The usage of relative sample counts leads to an overestimation of the resource demands. Estimation via absolute sample counts shows more promising but nevertheless, the issues raised above have to be addressed.

Details of experiment with ID 3 (JRA total sample count = 4468, `weblogic.work.ExecuteThread#run` sample count = 3565, effective processing time = $0.495 \cdot 1000 \cdot 1140$ sec = 564300 ms).						
Metric	**sWO_int_2**	**sWO_int_6**	**getInv_int_1**	**fA_int_1**	**cLO_int_2**	**Sum**
ic_{WLDF}	28721	286422	286422	85768	57047	-
Sample count (sc_{JRA})	426	86	919	504	33	1968
$\mathcal{D}_{\text{WLS\_CPU}}$ [ms] estimated via Service Demand Law: Partitioning with relative sample counts	4.25	0.09	0.92	1.69	0.17	-
$\mathcal{D}_{\text{WLS\_CPU}}$ [ms] estimated via Service Demand Law: Partitioning with absolute sample counts	2.35	0.05	0.51	0.93	0.09	-

Table 7.3: Resource demand estimations for Experiment 3 of Table 7.2.

$\mathcal{D}_{\text{WLS\_CPU}}$ [ms] estimated via:	Experiment ID	sWO_int_2	sWO_int_6	getInv_int_1	fA_int_1	cLO_int_2
Response times: $\mathcal{R}_{\text{WLDF}}$	1	3.28	0.04	0.96	1.04	0.05
Service Demand Law: Partitioning with relative sample counts	3	4.25	0.09	0.92	1.69	0.17
	4	5.12	0.07	0.83	1.60	0.15
	5	5.21	0.10	1.10	0.95	0.10
Service Demand Law: Partitioning with absolute sample counts	3	2.35	0.05	0.51	0.93	0.09
	4	2.92	0.04	0.47	0.91	0.09
	5	2.65	0.05	0.56	0.48	0.05

Table 7.4: Comparison of estimated resource demands.

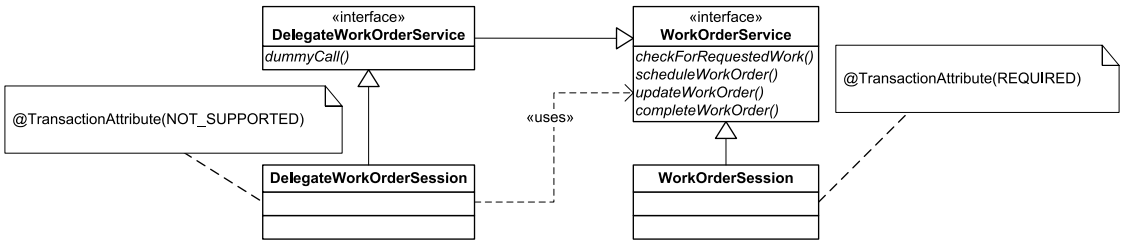


Figure 7.3: UML class diagram of the delegating EJB `DelegateWorkOrderSession`.

7.3.4 Summary

The evaluation of different monitoring approaches points out the challenges. Monitoring should impose a low overhead but provide steady results.

With regard to the results of Section 7.3.1 and Section 7.3.2, separating the extraction steps concerning the structure and resource demand extraction (see Section 6.2) is an appropriate design decision.

Regarding the method sampling approach evaluated in 7.3.3, further investigations are required. See the subsequent section for an evaluation of the other approach for resource demand estimation, namely partitioning the resource utilization via weighted response time ratios.

7.4 Case Study

With the tool prototype (see Chapter 6) we conduct a case study to evaluate the applicability of our approach. For the case study we use the benchmark application described in Section 7.1. We consider two scenarios. In the first scenario, the workload only consists of calls to business method `WorkOrderSession#scheduleWorkOrder`, i.e., we extract only those parts of the benchmark application that are required to process that method. In the second scenario, the workload consists of the benchmark operation `CreateVehicleEJB`. Thus, in this scenario we extract the main parts of the manufacturing domain.

As mentioned in Section 5.6.2.1, we approximate the DBS CPU demand by transactions's commit phases. To enable measuring of commit phase response times, we follow the approach described in Section 6.2.2. For the `WorkOrderSession` EJB we introduce a delegating stateless session bean annotated with transaction attribute `NOT_SUPPORTED`. Figure 7.3 illustrates the delegation. In the PCM, demands for the DBS CPU are then modelled as external service call to service `dbaccess(Double resourceDemands)`. We put that service in an `IJDBCDriver` interface which is provided by the component `JDBCDriver` we introduce. The RDSEFF of the component service then consists of one internal action with a CPU demand equal to the given input parameter value. Finally, the single assembly context of the `JDBCDriver` component is allocated to a DBS resource container.

Until now, we considered only demands of the application itself, i.e., CPU demands of internal actions. Resource demands for establishing connections are not considered yet. Since the benchmark driver accesses the EJBs via RMI we conducted an experiment to quantify RMI connection overhead at the WLS instance. We implemented a

Calls to business method `DelegateWorkOrderSession#dummyCall` (ramp up time = 600 sec, steady state time = 1140 sec, ramp down time = 60 sec).		
$\mathcal{U}_{\text{WLS\_CPU}}$	ic_{Driver}	$\mathcal{D}_{\text{WLS\_CPU}}$ [ms]
0.45	2277515	0.23

Table 7.5: Experiment to quantify RMI connection overhead at the WLS instance.

benchmark driver that calls the delegating EJB's business method `dummyCall`. The method body is empty. Since there are neither method arguments nor method return values, marshalling and demarshalling overhead is not considered. Table 7.5 shows the experimental results. The connection overhead is estimated with the Service Demand Law. A connection overhead of 0.23 ms is considered negligible because the case study's benchmark operations are measured with millisecond granularity.

Besides the average response times of benchmark operations, in this case study we consider the avg. utilization of the WLS CPU and the avg. utilization of the DBS CPU.

7.4.1 Scenario 1: Schedule Work Order

The benchmark driver is configured to only call business method `DelegateWorkOrderSession#scheduleWorkOrder` targeting a specific throughput. During driver run-time, we use the tool prototype to extract two performance models, one PCM instance for each implemented resource demand extraction approach. Thus, we have one PCM instance with resource demands estimated from the measured response times (Model A) and one PCM instance with resource demands estimated with weighted response time ratios and the Service Demand Law (Model B).

The resource demands for Model A are extracted during light system load ($\mathcal{U}_{\text{WLS\_CPU}}$ = 0.12, ramp up time = 120 sec, steady state time = 1020 sec). The resource demands for Model B are extracted during high system load ($\mathcal{U}_{\text{WLS\_CPU}}$ = 0.81, ramp up time = 120 sec, steady state time = 1020 sec). For validation we proceed as follows. On the one hand we simulate calls to `scheduleWorkOrder` using the two extracted PCM instances (measurement count = 100000). On the other hand, we conduct measurements of the real system, i.e., without any instrumentation. Then we compare the predicted CPU utilization and the predicted average response times with the actual measurements. The comparison is repeated for low load conditions ($\approx$ 20%), medium load conditions ($\approx$ 40% and $\approx$ 60%) and high load conditions ($\approx$ 80%).

Table 7.6 shows the results. Obviously, predictions based on Model B are better than predictions based on Model A. For a high throughput Model A delivers no performance predictions (indicated by "-") because the throughput cannot be achieved anymore. For Model B, the prediction error is mostly about 20%. In the experiment with the highest throughput, it is the overestimated WLS CPU utilization that causes the response time deviation of 77%. Both models underestimate the DBS CPU utilization while overestimating WLS CPU utilization. For Model B, we assume the overestimation to be caused by the instrumentation overhead during resource demand estimation. For Model A, we assume the overestimation to be caused by measuring also queue waiting times during resource demand estimation.

Throughput: 13.32 ops/sec					
Metric	**Measurement**	**Model A**	**Error**	**Model B**	**Error**
$\mathcal{U}_{\text{WLS\_CPU}}$	0.157	0.195	0.19	0.179	0.12
$\mathcal{U}_{\text{DBS\_CPU}}$	0.074	0.053	0.28	0.049	0.34
$\mathcal{R}_{\text{Driver}}$	19.4	22.4	0.14	20.4	0.05

Throughput: 33.63 ops/sec					
Metric	**Measurement**	**Model A**	**Error**	**Model B**	**Error**
$\mathcal{U}_{\text{WLS\_CPU}}$	0.377	0.492	0.23	0.454	0.17
$\mathcal{U}_{\text{DBS\_CPU}}$	0.161	0.134	0.17	0.124	0.23
$\mathcal{R}_{\text{Driver}}$	26.1	33.7	0.23	29.0	0.10

Throughput: 49.93 ops/sec					
Metric	**Measurement**	**Model A**	**Error**	**Model B**	**Error**
$\mathcal{U}_{\text{WLS\_CPU}}$	0.570	0.732	0.22	0.675	0.16
$\mathcal{U}_{\text{DBS\_CPU}}$	0.230	0.191	0.17	0.184	0.20
$\mathcal{R}_{\text{Driver}}$	35.5	59.7	0.41	46.3	0.23

Throughput: 71.12 ops/sec					
Metric	**Measurement**	**Model A**	**Error**	**Model B**	**Error**
$\mathcal{U}_{\text{WLS\_CPU}}$	0.851	-	-	0.962	0.12
$\mathcal{U}_{\text{DBS\_CPU}}$	0.314	-	-	0.262	0.17
$\mathcal{R}_{\text{Driver}}$	81.8	-	-	353.0	0.77

Table 7.6: Scenario 1: Validation of the `scheduleWorkOrder` performance model.

7.4.2 Scenario 2: Benchmark Operation Create Vehicle EJB

In this scenario we use the benchmark operation `CreateVehicleEJB` as workload generator. Using the tool prototype we extract two performance models for the manufacturing domain, one PCM instance for each resource demand extraction approach. Thus, we have one PCM instance with resource demands estimated from the measured response times (Model A) and one PCM instance with resource demands estimated with weighted response time ratios and the Service Demand Law (Model B). For the latter model, $\mathcal{U}_{\text{WLS\_CPU}}$ is partitioned using the weighted internal actions's response time ratios while $\mathcal{U}_{\text{DBS\_CPU}}$ is partitioned using the weighted commit phases's response time ratios.

The resource demands for Model A are extracted during light system load ($\mathcal{U}_{\text{WLS\_CPU}}$ = 0.09, ramp up time = 600 sec, steady state time = 1140 sec). The resource demands for Model B are extracted during high system load ($\mathcal{U}_{\text{WLS\_CPU}}$ = 0.75, ramp up time = 600 sec, steady state time = 1140 sec). For validation we proceed as follows. On the one hand we simulate the manufacturing domain using the two extracted PCM instances (measurement count = 100000). On the other hand, we conduct measurements of the real system, i.e., without instrumentation. Then we compare the predicted CPU utilization and the predicted average response times with the actual measurements. The comparison is repeated for low load conditions ($\approx$ 20%), medium load conditions ($\approx$ 40% and $\approx$ 60%) and high load conditions ($\approx$ 80%).

For simulation, we have to model the considered benchmark operation in a PCM usage model instance. Note that the considered benchmark operation calls four dif-

Throughput: 11.25 ops/sec					
Metric	**Measurement**	**Model A**	**Error**	**Model B**	**Error**
$\mathcal{U}_{\text{WLS\_CPU}}$	0.188	0.234	0.20	0.234	0.20
$\mathcal{U}_{\text{DBS\_CPU}}$	0.112	0.133	0.16	0.086	0.23
$\mathcal{R}_{\text{corr\_avg}}$	39.9	41.0	0.03	35.7	0.11

Throughput: 22.21 ops/sec					
Metric	**Measurement**	**Model A**	**Error**	**Model B**	**Error**
$\mathcal{U}_{\text{WLS\_CPU}}$	0.422	0.463	0.09	0.463	0.09
$\mathcal{U}_{\text{DBS\_CPU}}$	0.180	0.263	0.32	0.129	0.28
$\mathcal{R}_{\text{corr\_avg}}$	50.6	55.1	0.08	48.3	0.05

Throughput: 33.90 ops/sec					
Metric	**Measurement**	**Model A**	**Error**	**Model B**	**Error**
$\mathcal{U}_{\text{WLS\_CPU}}$	0.606	0.711	0.15	0.705	0.14
$\mathcal{U}_{\text{DBS\_CPU}}$	0.275	0.403	0.32	0.258	0.06
$\mathcal{R}_{\text{corr\_avg}}$	72.3	91.4	0.21	81.9	0.12

Throughput: 43.60 ops/sec					
Metric	**Measurement**	**Model A**	**Error**	**Model B**	**Error**
$\mathcal{U}_{\text{WLS\_CPU}}$	0.788	0.906	0.13	0.909	0.13
$\mathcal{U}_{\text{DBS\_CPU}}$	0.335	0.524	0.36	0.333	0.01
$\mathcal{R}_{\text{corr\_avg}}$	128.8	250.1	0.49	244.7	0.47

Table 7.7: Scenario 2a: Validation of the `CreateVehicleEJB` performance model.

ferent business methods in a specific order and contains three pauses implemented by calling `Thread.sleep(333)`. That is why we do not compare the actual benchmark operation's response time but the corrected response time $\mathcal{R}_{\text{corr\_avg}}$. We denote a response time as corrected if the length of enclosed delay intervals is subtracted. While SimuCom simulates exact delays of 333 ms, the benchmark driver calls to `Thread.sleep(333)` not always guarantee a delay of 333 ms [**?**]. Thus, we have to adapt the benchmark driver to measure the corrected response time instead of the overall response time.

In the appendix in Section B.2, there are screenshots showing the corresponding PCM usage model and models extracted by the tool prototype.

Table 7.7 shows the results for the manufacturing domain. Generally, predictions based on Model B are better than predictions based on Model A. For Model B, the prediction error is mostly about 20%. In the experiment with the highest throughput, it is the overestimated WLS CPU utilization that causes the response time deviation of 47%. For high CPU utilization like 80% an overestimated utilization results in a considerably higher response time. Both models overestimate the WLS CPU utilization. Model A also overestimates the DBS CPU utilization while this is underestimated by Model B. Similar to the previous scenario, we assume the overestimation of Model B to be caused by the instrumentation overhead during resource demand estimation.

Table 7.8 shows simulation results where the performance model of the manufacturing domain is used to predict the performance of only the `scheduleWorkOrder` method. The difference between these simulations and the simulations conducted

Throughput: 13.32 ops/sec					
Metric	**Measurement**	**Model A**	**Error**	**Model B**	**Error**
$\mathcal{U}_{\text{WLS\_CPU}}$	0.157	0.186	0.16	0.189	0.17
$\mathcal{U}_{\text{DBS\_CPU}}$	0.074	0.055	0.26	0.031	0.58
$\mathcal{R}_{\text{Driver}}$	19.4	21.6	0.10	20.1	0.04

Throughput: 33.63 ops/sec					
Metric	**Measurement**	**Model A**	**Error**	**Model B**	**Error**
$\mathcal{U}_{\text{WLS\_CPU}}$	0.377	0.468	0.19	0.478	0.21
$\mathcal{U}_{\text{DBS\_CPU}}$	0.161	0.140	0.13	0.079	0.51
$\mathcal{R}_{\text{Driver}}$	26.1	31.3	0.16	29.9	0.12

Throughput: 49.93 ops/sec					
Metric	**Measurement**	**Model A**	**Error**	**Model B**	**Error**
$\mathcal{U}_{\text{WLS\_CPU}}$	0.570	0.695	0.18	0.713	0.20
$\mathcal{U}_{\text{DBS\_CPU}}$	0.230	0.208	0.10	0.119	0.48
$\mathcal{R}_{\text{Driver}}$	35.5	50.9	0.30	51.9	0.32

Throughput: 71.12 ops/sec					
Metric	**Measurement**	**Model A**	**Error**	**Model B**	**Error**
$\mathcal{U}_{\text{WLS\_CPU}}$	0.851	-	-	-	-
$\mathcal{U}_{\text{DBS\_CPU}}$	0.314	-	-	-	-
$\mathcal{R}_{\text{Driver}}$	81.8	-	-	-	-

Table 7.8: Scenario 2b: Validation of the `CreateVehicleEJB` performance model.

for the scenario in Section 7.4.1 (see Table 7.6) is that the underlying performance models are extracted using a different transaction mix. As expected, this section's Model A performs similar to Model A of the previous section. The performance predictions based on Model B do not match the measurements well. While here, predictions for the WLS CPU utilization are about 5% higher than the predictions of the previous section, the DBS CPU utilization is seriously underestimated. This leads to the conclusion that partitioning via weighted response time ratios appears to be appropriate for WLS CPU demands but not for DBS CPU demands. In other words, partitioning the DBS CPU demands by the commit phase response time ratios has to be reconsidered.

7.5 Summary

In this chapter we evaluated different monitoring approaches and presented a case study evaluating the accuracy of the extracted performance models. As the simulation results show, the extraction method delivers appropriate performance models. However, there are still issues to be addressed. The estimated resource demands should be calibrated with regard to the instrumentation overhead. While the estimated WLS resource demands are already adequate, the estimation of DBS resource demands has to be improved.

Furthermore, there are some technical issues. Performance models should be able to consider dynamic frequency scaling technologies like EIST. Additionally, the PCM Bench should provide simulations for multi-core CPUs. Moreover, the overhead of

monitoring tools plays a crucial role when extracting performance models during operation.

8. Conclusion

This thesis concludes with a summary of preceding chapters and a section on future work.

8.1 Summary

In this thesis we developed a method for semi-automated extraction of Palladio Component Model (PCM) instances of Java EE applications based on monitoring data collected during operation. The method is implemented in a proof-of-concept tool prototype. To obtain monitoring data we use state-of-the-art, industrial monitoring tools available for the current version of the Oracle WebLogic Server (WLS) Platform. Namely, these are the WebLogic Diagnostics Framework (WLDF) and the JRockit Runtime Analyzer (JRA). In the context of a case study we evaluated our approach with a real-world enterprise application. For reasons of external validity we chose a beta version of the successor of the SPECjAppServer2004 benchmark application.

Extracting a PCM instance requires the extraction of the application's architecture, the application's performance-relevant behavior and the application's resource demands. We extracted structural information by means of call path tracing enabled by the WLDF instrumentation engine. For resource demand extraction we presented several approaches. Basically, resource demands are estimated either with response times or with the Service Demand Law.

We focused on the EJB 3.0 Component Model, the web tier or user interface technologies like Java Server Faces are not considered. As persistence framework we considered JPA. The output of our extraction method is a PCM (component) repository model and a PCM system model. The former describes the application's components, their behavior and their relationships with each other. The system model shows the components as they are actually deployed. The extraction assumes information about the component boundaries to be available. Furthermore, for the current prototype implementation component code has to be structured in a form that makes performance-relevant intra-component control flow explicit. This is required due to the lack of support for in-method instrumentation in current monitoring tools.

Given that the extraction is based on trace data the method may not extract the entire application but only those parts that are actually called during monitoring. Thus, we model the effective architecture of the considered application.

The method as it is implemented in the tool prototype separates the extraction approach into structure extraction and resource demand extraction. We assume the considered application to be running on a WLS instance. The extraction can be started as soon as the prototype is provided with information on the component boundaries. To introduce application monitors the application of interest has to be redeployed unless the diagnostic actions are already injected and only disabled. In the latter case, enabling instrumentation can be performed without a redeployment. Note that the proof-of-concept prototype does not configure WLDF automatically. However, once the application is instrumented it has to be exposed to workload in order to raise gathering of trace data. Since the overhead of trace data generation cannot be neglected, the workload intensity should not be high in this step. The tool's user decides when to stop the structure extraction and when to start the time interval for resource demand extraction. For the extraction of resource demands we use low overhead instrumentation. Here, the system performance is affected to a degree we consider acceptable. The way resource demands should be estimated depends on the system load. We implemented two approaches, one for light load and one for medium to heavy load.

We evaluated the applicability of the method on a part of the beta version of the successor of the SPECjAppServer2004 benchmark which we deployed in a realistic system environment. We extracted performance models and conducted performance predictions which we compared to actual measurements. Mostly, we encountered an error of about 20 to 30%. However, further calibrations of estimated resource demands are possible.

The described method automates extraction of PCM instances from Java EE applications. Even though some intervention is required, the prototype we implemented provides a proof-of-concept showing how the existing gap between low level monitoring data and high level performance models can be closed. It also reveals issues needed to solve for a complete automation.

As part of the thesis, a collaboration with Oracle was conducted. Results of this thesis will be published on the Oracle Technology Network. In addition, we plan to submit a research paper to the first international workshop on run-time models for self-managing systems and application (ROSSA 2009).

8.2 Future Work

The considered Java EE benchmark can be used as a basis for future work. To capture not only a part of the benchmark but the entire application, the scope with regard to the covered Java EE technologies has to be extended:

- The extraction tool will be extended to extract also message-driven beans and asynchronous message flows. To consider not only point-to-point but also publish-subscribe messaging, i.e., general JMS messaging, the PCM has to be adapted too. Currently, the PCM does not allow for modelling message-oriented-middleware mechanisms.

- Beyond EJB 3.0, the web tier respectively Java Servlets will be considered to completely model the benchmark.
- Our extraction method currently does not distinguish between stateless session beans and stateful session beans. It is unlikely that the performance of a stateful bean can be accurately predicted if its state is not considered.

In addition, the extraction tool can be enhanced concerning the following aspects:

- We will provide a GUI to ease the usage of the tool. An automated WLDF configuration would also ease application of the extraction method. For that purpose, WLDF configuration would have to be simplified. Currently, the provided monitor management functionalities are limited.
- The tool can be enhanced to support model extraction of applications that are deployed on more than one WLS instance.
- Another issue that requires further considerations is that the currently needed instrumentation enforces a redeployment of the application. Given that the approach aims for PCM instance extraction of running Java EE applications this is an important challenge.
- Given that the monitoring tools provide detailed information on workload and resource utilization data, one can integrate the data into the PCM. Event though such information cannot be used as input data for the performance model it would be useful to detect hot spots or bottlenecks.
- In the current version, the extraction tool does not generate a PCM resource environment or a PCM allocation model. In future versions, the tool should support these models as well. In addition, an automated extraction of a PCM usage model based on monitoring data is imaginable.
- The implementation can be extended concerning the extraction of probabilistic parameter dependencies. However, automated extraction of performance-relevant behavior and modelling of PCM RDSEFFS remains an issue. For the proof-of-concept prototype we implemented a workaround which should be revised to be applicable in practice. Thus, further research on how to overcome the workaround of refactoring component code is needed. Therefore, the work presented in [**?**] is of great interest.

Beyond, the PCM Bench should provide support to simulate (hardware-implemented) techniques for power reduction like dynamic frequency scaling. Otherwise, the prediction accuracy will suffer unless these techniques are disabled. Especially if the performance predictions aim for run-time performance management with the objective to reduce power consumption, deactivation can not be a solution.

In general, the monitoring tools should impose an overhead as low as possible. The influence on the system and application should be minimized in order to not cause failing performance objectives on the one hand and to extract representative monitoring data on the other hand. The latter requirement particularly holds for the extraction of resource demands. One approach to reduce the overall monitoring

overhead would be to implement an adaptive monitoring that allows monitoring for sensitive parts of the application. In conjunction with the monitoring approaches, further investigations on resource demand estimation are desirable. To give a concrete example, the usefulness of method sampling data from the JRockit Runtime Analyzer needs some further investigation.

A. Tool Prototype Implementation Details

A.1 XML Schema for System Description

```
<?xml version="1.0" encoding="UTF-8"?>

<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"
           targetNamespace="http://sdq.ipd.uni-karlsruhe.de/fabro"
           elementFormDefault="qualified"
           attributeFormDefault="unqualified">

  <xs:complexType name="rootType">
    <xs:annotation><xs:documentation>
      The root type for all types defined in this schema.
    </xs:documentation></xs:annotation>
    <xs:sequence>
      <xs:element name="name" type="xs:string" minOccurs="0" nillable="true"/>
    </xs:sequence>
  </xs:complexType>

  <xs:simpleType name="application-name-type">
    <xs:annotation><xs:documentation>
      The type representing the application name of the application to select.
    </xs:documentation></xs:annotation>
    <xs:restriction base="xs:string"></xs:restriction>
  </xs:simpleType>

  <xs:simpleType name="configmode-type">
    <xs:annotation><xs:documentation>
      The type representing the configmode. The configmode enables options
      of how to map EJBs to Components. Either each EJB is mapped
      to its own component, or a component consists of all EJBs
      contained in the same Java package, or the user manually
      maps EJBs and components.
    </xs:documentation></xs:annotation>
    <xs:restriction base="xs:string">
      <xs:enumeration value="ejbtocomponent"/>
      <xs:enumeration value="packagetocomponent"/>
      <xs:enumeration value="manual"/>
    </xs:restriction>
  </xs:simpleType>

  <xs:complexType name="component-type">
    <xs:annotation><xs:documentation>
      The type representing a component specification.
      A component has a name and more than one EJBs contained.
    </xs:documentation></xs:annotation>
    <xs:sequence>
```

```
      <xs:element name="name" type="xs:string" maxOccurs="1"
                  minOccurs="1" nillable="false"/>
      <xs:element name="ejb" maxOccurs="unbounded" type="xs:string"
                  minOccurs="1" nillable="false"/>
    </xs:sequence>
  </xs:complexType>

  <xs:complexType name="system-type">
    <xs:annotation><xs:documentation>
      The type representing the root element of the system description XML file.
      In addition to the application's name, the configmode
      and the component specifications, the include and exclude elements
      can be used to filter the classes that should be considered.
    </xs:documentation></xs:annotation>
    <xs:complexContent>
      <xs:extension xmlns:sys="http://sdq.ipd.uni-karlsruhe.de/fabro"
                    base="sys:rootType">
        <xs:sequence>
          <xs:element name="application" type="sys:application-name-type"
                      nillable="false" maxOccurs="1" minOccurs="1"/>
          <xs:element name="configmode" type="sys:configmode-type"
                      maxOccurs="1" minOccurs="1" nillable="false"/>
          <xs:element name="include" maxOccurs="unbounded"
                     type="xs:string" minOccurs="0" nillable="false"/>
          <xs:element name="exclude" maxOccurs="unbounded"
                      type="xs:string" minOccurs="0" nillable="false"/>
          <xs:element name="component" type="sys:component-type"
                      maxOccurs="unbounded" minOccurs="0" nillable="false"/>
        </xs:sequence>
      </xs:extension>
    </xs:complexContent>
  </xs:complexType>

  <xs:element xmlns:sys="http://sdq.ipd.uni-karlsruhe.de/fabro"
              name="system" type="sys:system-type">
    <xs:annotation><xs:documentation>
      A component name has to be unique amongst the other component names.
      An EJB may be mapped to at most one component.
    </xs:documentation></xs:annotation>
    <xs:unique name="unique-component-name">
      <xs:selector xpath="component"/>
      <xs:field xpath="name"/>
    </xs:unique>
    <xs:unique name="unique-component-ejb">
      <xs:selector xpath="component"/>
      <xs:field xpath="ejb"/>
    </xs:unique>
  </xs:element>

</xs:schema>
```

A.2 Design of Prototype Implementation

The tool prototype is implemented as a Eclipse plug-in in Java. In order to get an overview on how the implementation is organized, Figure A.1 shows the package structure. Only the main packages and their dependencies are shown.

The internal structure of the six mentioned packages is illustrated in Figures A.2, A.3, A.4, A.6, A.7 and A.5. Subpackages are not shown in detail.

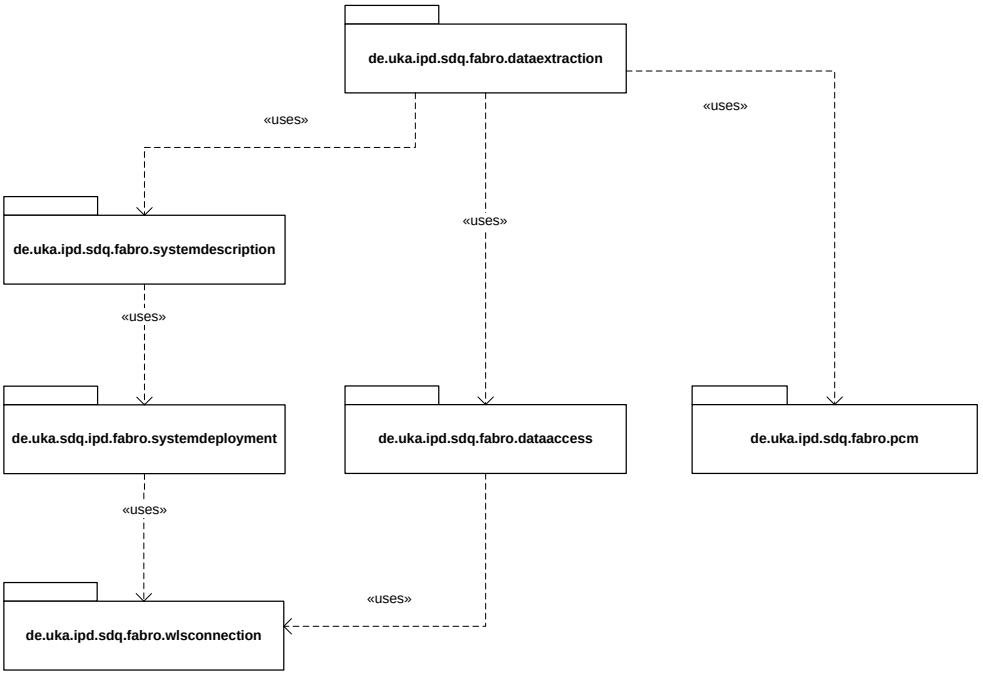


Figure A.1: Package overview of the tool prototype.

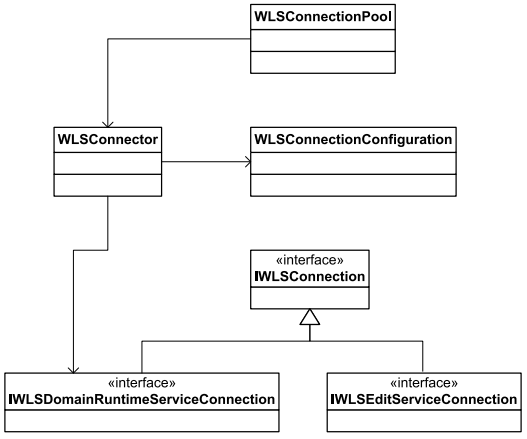


Figure A.2: Overview of `de.uka.ipd.sdq.fabro.wlsconnection`.

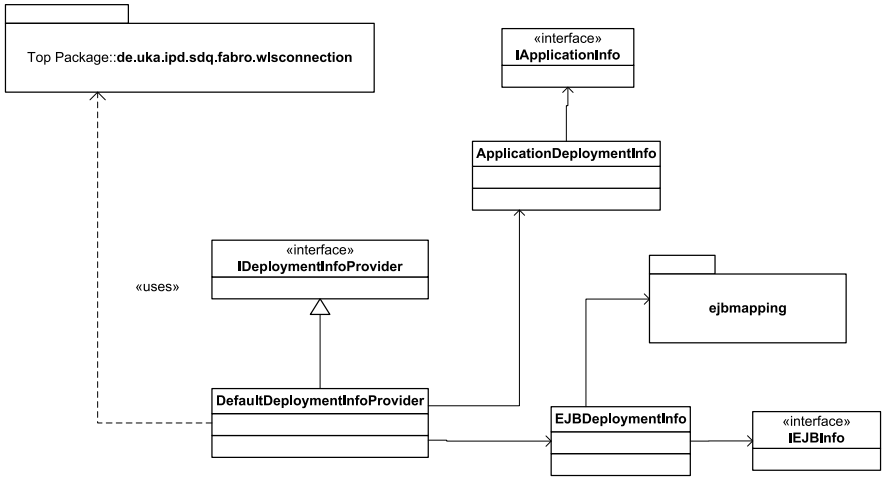


Figure A.3: Overview of `de.uka.ipd.sdq.fabro.systemdeployment`.

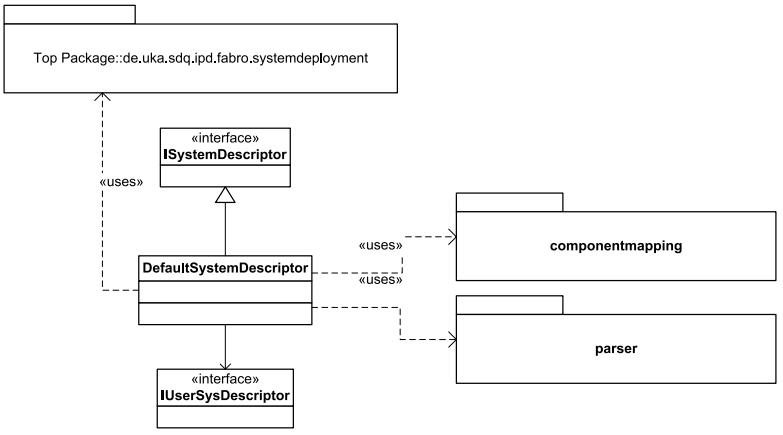


Figure A.4: Overview of `de.uka.ipd.sdq.fabro.systemdescription`.

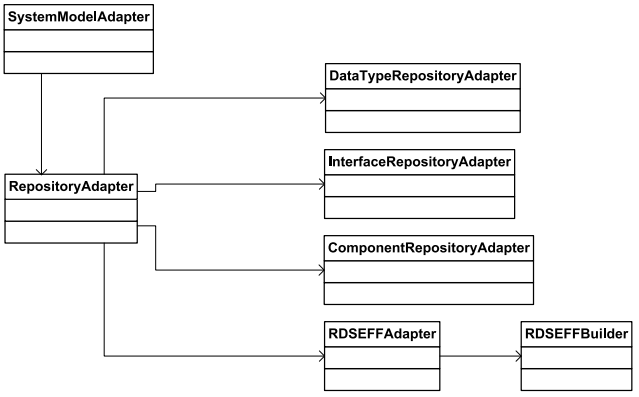


Figure A.5: Overview of `de.uka.ipd.sdq.fabro.pcm`.

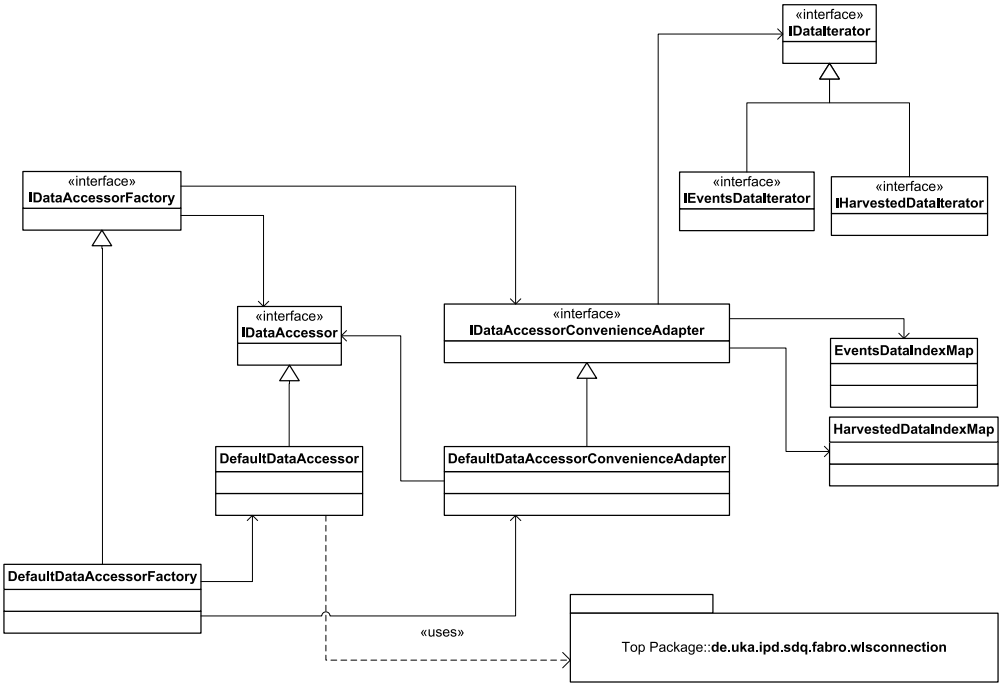


Figure A.6: Overview of `de.uka.ipd.sdq.fabro.dataaccess`.

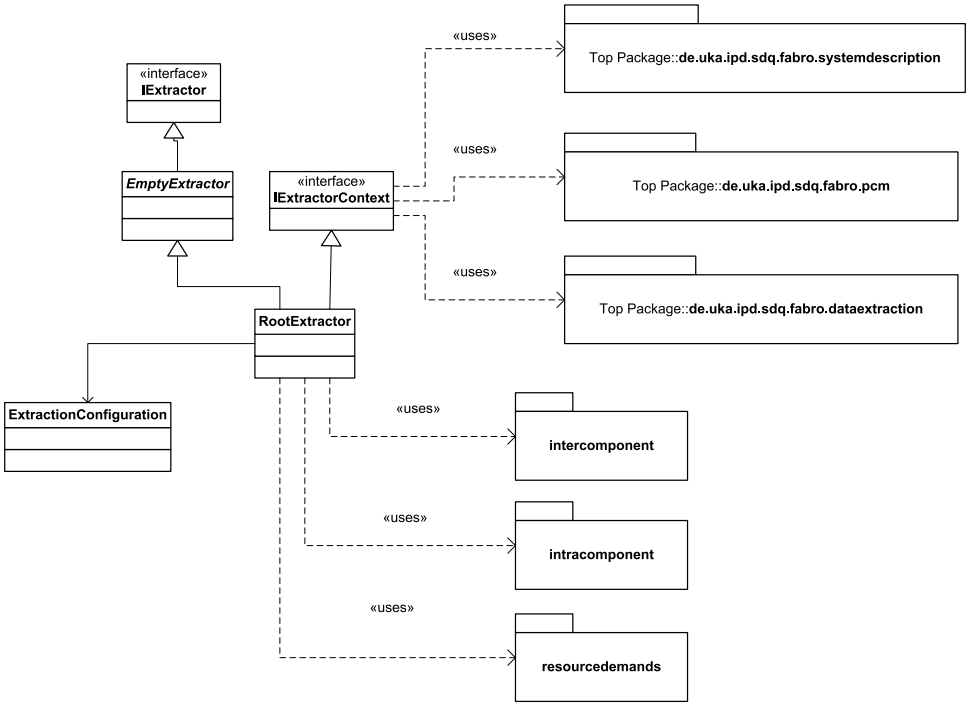


Figure A.7: Overview of `de.uka.ipd.sdq.fabro.extraction`.

B. Evaluation Details

B.1 Instrumentation versus JRockit Optimization

We would like to find out if an injected action can be optimized away by JRockit. Therefore, we instrument different methods with different actions. Then we run a specific driver that logs method call counts. Comparing the method call counts obtained through instrumentation and the method call counts logged by the driver we can deduce if an action was optimized away.

We execute a transaction mix of the two business methods `scheduleWorkOrder` (sWO) and `createLargeOrder` (cLO). Both business methods trigger internal actions that are denoted as follows:

sWO_int_2:	`WorkOrderSession#scheduleWorkOrder_1_internalaction_2`
sWO_int_6:	`WorkOrderSession#scheduleWorkOrder_1_internalaction_6`
getInv_int_1:	`MfgSession#getInventory_1_internalaction_1`
fA_int_1:	`MfgSession#findAssembly_1_internalaction_1`
cLO_int_2:	`LargeOrderSession#createLargeOrder_1_internalaction_2`

Note that getInv_int_1 is only called by sWO whereas fA_int_1 is called by both sWO and cLO.

Four experiment runs are conducted. Table B.1 shows the experimental results. In Experiment 1, we instrumented with the `call` pointcut expression. We instrumented sWO with diagnostic action `TraceElapsedTimeAction` and cLO with action `MethodInvocationStatisticsAction`. Obviously, the expected method invocation counts differ from the measured counts of sWO but match with the measured counts of cLO. Notice that all instrumented methods are also marked as optimized in JRA (indicated by the asterisk *).

Figure B.1 shows a screenshot of a JRockit Runtime Analyzer (JRA) recording. Methods annotated with a "lightning" have been optimized. For instance, the screenshot shows that method sWO has been optimized.

In Experiment 2, we instrumented with the `execution` pointcut expression and received similar results as for Experiment 1. The kind of the pointcut expression appears not to have an influence on code optimization.

1	**Pointcut specification**	`call(...)`							
	Diagnostic action	`TraceElapsedTimeAction`					`MethodInvocationStatisticsAction`		
	Instrumented method	sWO*	sWO_int_2*	sWO_int_6*	fA_int_1*	getInv_int_1*	cLO*	cLO_int_2*	fA_int_1*
	Expected count	8014	8014	$\geq$ 8014	8014+24767=32781	$\geq$ 8014	24767	24767	8014+24767=32781
	Monitored count	190	190	1923	726	1958	24767	24767	32781

2	**Pointcut specification**	`execution(...)`							
	Diagnostic action	`TraceElapsedTimeAction`					`MethodInvocationStatisticsAction`		
	Instrumented method	sWO*	sWO_int_2*	sWO_int_6*	fA_int_1*	getInv_int_1*	cLO*	cLO_int_2*	fA_int_1*
	Expected count	8052	8052	$\geq$ 8052	8052+23655=31707	$\geq$ 8052	23655	23655	8052+23655=31707
	Monitored count	83	164	1666	324	833	23655	23655	31707

3	**Pointcut specification**	`execution(...)`							
	Diagnostic action	`MethodInvocationStatisticsAction`					`MethodInvocationStatisticsAction`		
	Instrumented method	sWO*	sWO_int_2*	sWO_int_6*	fA_int_1*	getInv_int_1*	cLO*	cLO_int_2*	fA_int_1*
	Expected count	7978	7978	$\geq$ 7978	7978+24840=32818	$\geq$ 7978	24840	24840	7978+24840=32818
	Monitored count	7978	7978	79089	32818	79089	24840	24840	32818

4	**JRockit's *method optimization* switched off for classes: LargeOrder, WorkOrderSession, MfgSession**								
	Pointcut specification	`execution(...)`							
	Diagnostic action	`TraceElapsedTimeAction`					`MethodInvocationStatisticsAction`		
	Instrumented method	sWO	sWO_int_2	sWO_int_6	fA_int_1	getInv_int_1	cLO	cLO_int_2	fA_int_1
	Expected count	6410	6410	$\geq$ 6410	6410+18739=25149	$\geq$ 6410	18739	18739	6410+18739=25149
	Monitored count	6410	6410	64584	25149	64584	18739	18739	32818

Legend:

cLO	LargeOrderSession#createLargeOrder	sWO	WorkOrderSession#scheduleWorkOrder
cLO_int_2	LargeOrderSession#createLargeOrder_1_internalaction_2	sWO_int_2	WorkOrderSession#scheduleWorkOrder_1_internalaction_2
getInv_int_1	MfgSession#getInventory_1_internalaction_1	sWO_int_6	WorkOrderSession#scheduleWorkOrder_1_internalaction_6
fA_int_1	MfgSession#findAssembly_1_internalaction_1	<method>*	method or a predecessor marked as optimized (in JRA)

Table B.1: Experiments to discover the influence of JRockit optimizations on instrumentation.

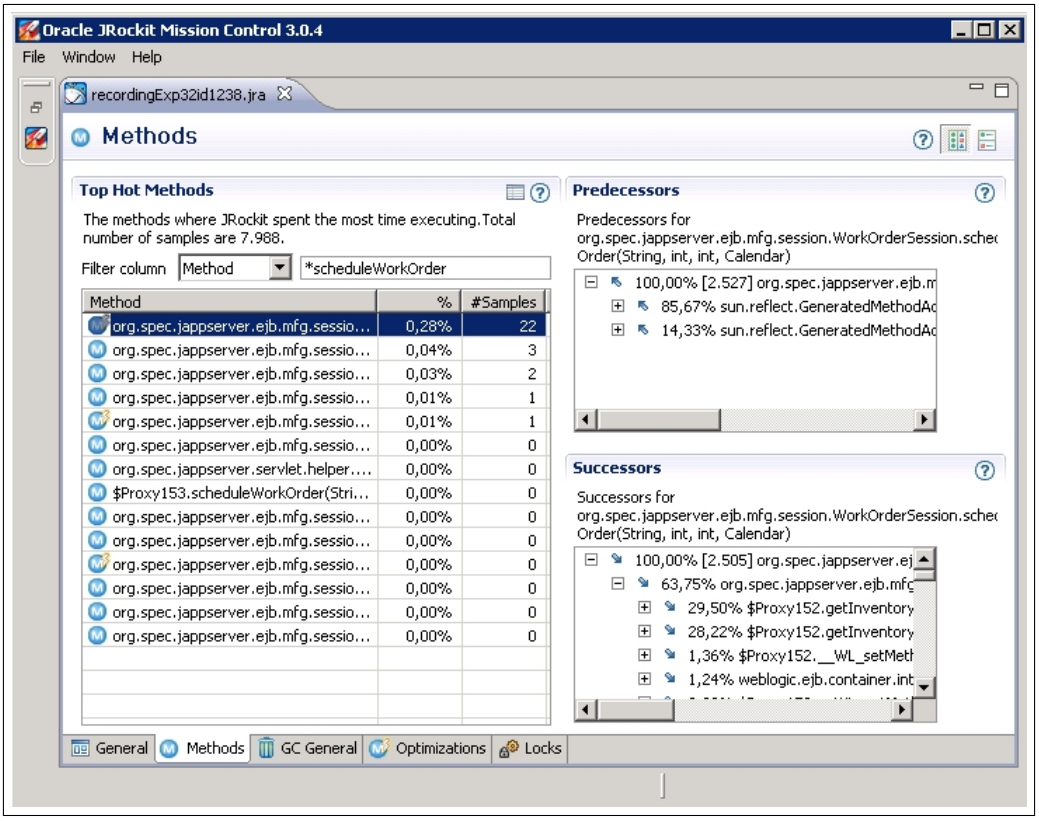


Figure B.1: Screenshot of JRockit Runtime Analyzer recording.

In Experiment 3, we instrumented both methods sWO and cLO with the action `MethodInvocationStatisticsAction`. There, for all instrumented methods the expected method invocation counts match with the measured invocation counts. For Experiment 4 we configured WLDF as in Experiment 2 but switched JRockit optimizations off. Obviously, this had an influence on the measured invocation counts.

To summarize, action `TraceElapsedTimeAction` appears to be affected by JRockit optimizations while `MethodInvocationStatisticsAction` is not. Nevertheless, this issue needs further investigation.

B.2 PCM Model Instances used in the Case Study

In this section we show screenshots of PCM model instances used for Scenario 2. Figure B.2 shows the output of the tool prototype, Figure B.3 shows the PCM usage model instance that is modelled manually.

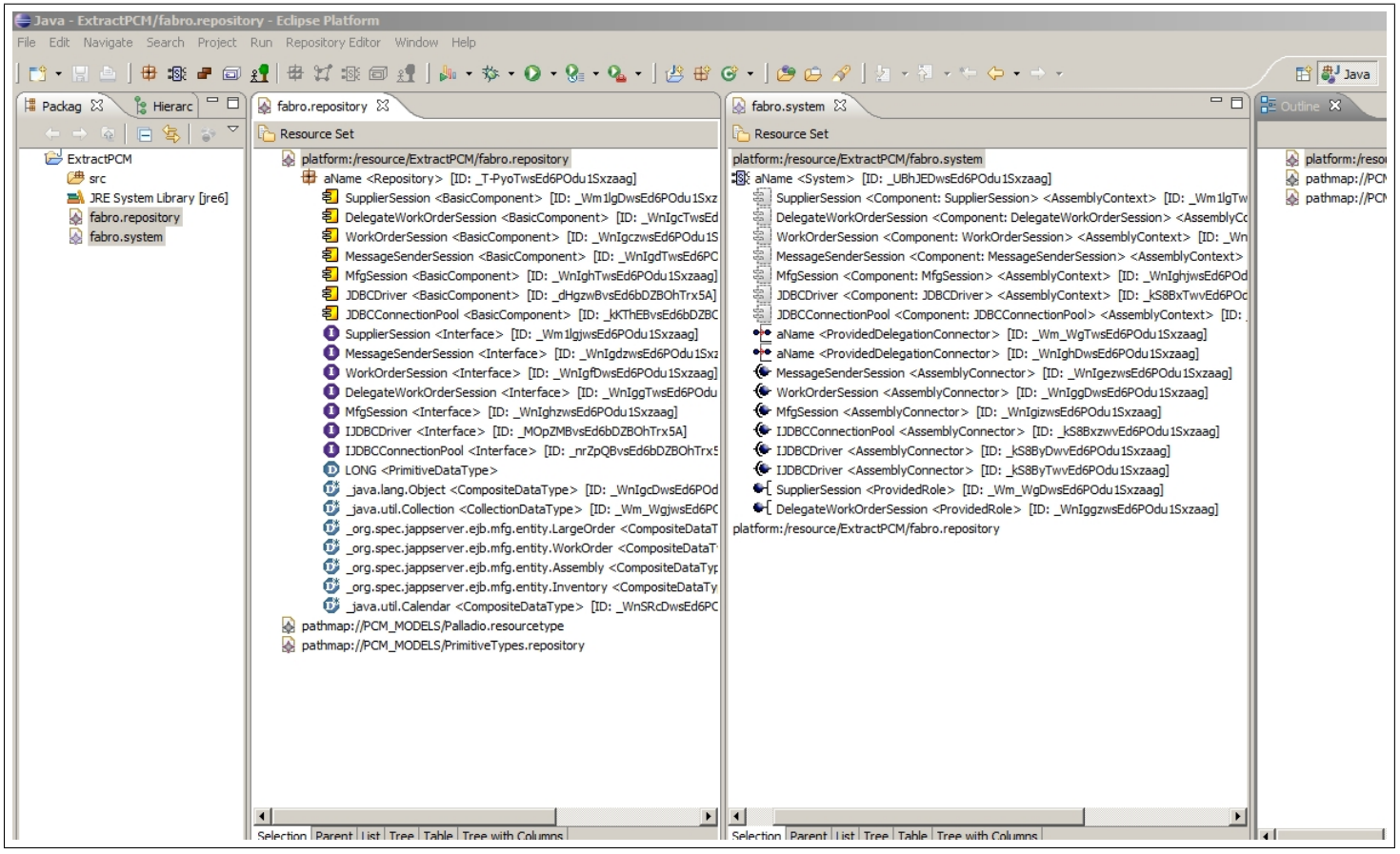


Figure B.2: Screenshot of a PCM repository model instance and a PCM system model instance as they are extracted by the tool prototype for Scenario 2a of the case study in Section7.4.2.

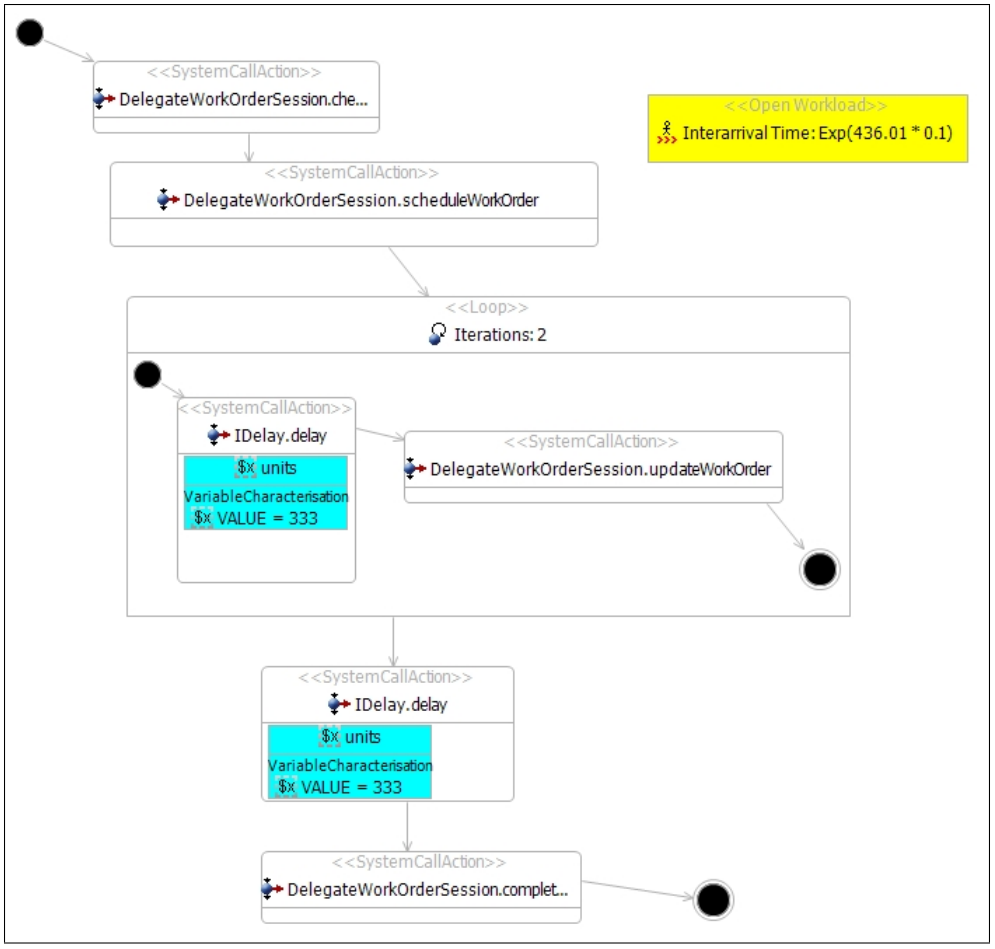


Figure B.3: Screenshot of the PCM usage model instance that is manually modelled for Scenario 2a of the case study in Section7.4.2.

Bibliography

[Bec08] Steffen Becker. *Coupled Model Transformations for QoS Enabled Component-Based Software Design*. Universitätsverlag Karlsruhe, 2008.

[Bed09] Dmitry Bedrin. jTracert. Online, 2009. Java agent implementation. http://code.google.com/p/jtracert/. Last visit: 2009-05-19.

[BKR07] Steffen Becker, Heiko Koziolek, and Ralf Reussner. Model-Based Performance Prediction with the Palladio Component Model. In *WOSP '07: Proceedings of the 6th international workshop on Software and performance*, pages 54–65, New York, NY, USA, 2007. ACM.

[BKR09] Steffen Becker, Heiko Koziolek, and Ralf Reussner. The Palladio Component Model for Model-Driven Performance Prediction. *Journal of Systems and Software*, 82:3–22, 2009.

[BLL06] Lionel C. Briand, Yvan Labiche, and Johanne Leduc. Toward the Reverse Engineering of UML Sequence Diagrams for Distributed Java Software. *IEEE Transactions on Software Engineering*, 32(9):642–663, September 2006.

[Car08] David Carrera. *Adaptive Execution Environments for Application Servers*. PhD thesis, Universitat Politècnica de Catalunya, Spain, 2008.

[CD00] John Cheesman and John Daniels. *UML Components: A Simple Process for Specifying Component-Based Software*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2000.

[CGT+03] David Carrera, Jordi Guitart, Jordi Torres, Eduard Ayguade, and Jesus Labarta. Complete instrumentation requirements for performance analysis of Web based technologies. In *ISPASS '03: Proceedings of the 2003 IEEE International Symposium on Performance Analysis of Systems and Software*, pages 166–175, Washington, DC, USA, 2003. IEEE Computer Society.

[Cho07] Landry Chouambe. Rekonstruktion von Software-Architekturen. Master's thesis, Universität Karlsruhe (TH), Germany, May 2007.

[CKK08] Landry Chouambe, Benjamin Klatt, and Klaus Krogmann. Reverse Engineering Software-Models of Component-Based Systems. In Kostas Kontogiannis, Christos Tjortjis, and Andreas Winter, editors, *12th European Conference on Software Maintenance and Reengineering*, pages 93–102, Athens, Greece, April1–4 2008. IEEE Computer Society.

[CLGL05] Shiping Chen, Yan Liu, Ian Gorton, and Anna Liu. Performance prediction of Component-Based Applications. *J. Syst. Softw.*, 74(1):35–43, 2005.

[DB78] Peter J. Denning and Jeffrey P. Buzen. The Operational Analysis of Queueing Network Models. *ACM Computing Surveys*, 10(3):225–261, 1978.

[DGL06] Marcus Denker, Orla Greevy, and Michele Lanza. Higher Abstractions for Dynamic Analysis. In *Proceedings of the 2nd International Workshop on Program Comprehension through Dynamic Analysis (PCODA 2006)*, pages 32–38, 2006.

[DPE04] Giovanni Denaro, Andrea Polini, and Wolfgang Emmerich. Early Performance Testing of Distributed Software Applications. In *WOSP '04: Proceedings of the 4th international workshop on Software and performance*, pages 94–103, New York, NY, USA, 2004. ACM.

[dyn] dynaTrace software. Tracing transactions across distributed Java/.NET application components at production-save overhead. Online. http://www.dynatrace.com/en/whitepapers_articles.aspx. Last visit: 2009-05-19.

[EFH04] Evgeni M. Eskenazi, Alexandre V. Fioukov, and Dieter K. Hammer. Performance Prediction for Component Compositions. In *Proceedings of the 7th International Symposium on Component-based Software Engineering (CBSE7)*, pages 280–293, 2004.

[FBB+99] Martin Fowler, Kent Beck, John Brant, William F. Opdyke, and Don Roberts. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley Professional, June 1999.

[Fou] The Eclipse Foundation. The AspectJ Project. Project homepage. http://eclipse.org/aspectj. Last visit: 2009-05-19.

[FP02] Mark Friedman and Odysseas Pentakalos. *Windows 2000 Performance Guide: Help for Windows 2000 Administrators and Application Developers*. O'Reilly & Associates, Inc., Sebastopol, CA, USA, 2002.

[Gla98] Robert L. Glass. *Software Runaways. Lessons learned from Massive Software Project Failures.* Prentice Hall, 1998.

[HWRI99] Curtis E. Hrischuk, Murray Woodside, Jerome A. Rolia, and Rod Iversen. Trace-Based Load Characterization for Generating Performance Software Models. *IEEE Transactions on Software Engineering*, 25(1):122–135, 1999.

[IWF07] Tauseef Israr, Murray Woodside, and Greg Franks. Interaction tree algorithms to extract effective architecture and layered performance models from traces. *J. Syst. Softw.*, 80(4):474–492, 2007.

[JBC+06] Eric Jendrock, Jennifer Ball, Debbie Carson, Ian Evans, Scott Fordin, and Kim Haase. *Java(TM) EE 5 Tutorial, The (3rd Edition) (The Java Series)*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 2006.

[KB03a] Samuel Kounev and Alejandro Buchmann. Performance Modeling and Evaluation of Large-Scale J2EE Applications. In *Proceedings of the 29th International Conference of the Computer Measurement Group˜(CMG) on Resource Management and Performance Evaluation of Enterprise Computing Systems˜- CMG2003, Dallas, TX, USA, December 7-12*, 2003.

[KB03b] Samuel Kounev and Alejandro Buchmann. Performance Modeling of Distributed E-Business Applications using Queueing Petri Nets. In *Proceedings of the 2003 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS 2003), Austin, Texas, USA, March 6-8, 2003*, pages 143–155, Washington, DC, USA, 2003. IEEE Computer Society.

[KBH07] Heiko Koziolek, Steffen Becker, and Jens Happe. Predicting the Performance of Component-Based Software Architectures with Different Usage Profiles. In *Proc. 3rd International Conference on the Quality of Software Architectures (QoSA'07)*, volume 4880 of *LNCS*, pages 145–163. Springer, July 2007.

[KHB06] Heiko Koziolek, Jens Happe, and Steffen Becker. Parameter Dependent Performance Specifications of Software Components. In Christine Hofmeister, Ivica Crnkovic, Ralf Reussner, and Steffen Becker, editors, *Proc. 2nd International Conference on the Quality of Software Architectures (QoSA'06)*, volume 4214 of *LNCS*, pages 163–179. Springer, June 2006.

[KKKR08] Thomas Kappler, Heiko Koziolek, Klaus Krogmann, and Ralf Reussner. Towards Automatic Construction of Reusable Prediction Models for Component-Based Performance Engineering. In Korbinian Herrmann and Bernd Brügge, editors, *Software Engineering 2008*, volume 121 of *LNI*, pages 140–154, Munich, Germany, February 2008. Bonner Köllen Verlag.

[KKR08a] Klaus Krogmann, Michael Kuperberg, and Ralf Reussner. Reverse Engineering of Parametric Behavioural Service Performance Models from Black-Box Components. In Ulrike Steffens, Jan Stefan Addicks, and Niels Streekmann, editors, *MDD, SOA und IT-Management (MSI 2008)*, pages 57–71, Oldenburg, September 2008. GITO Verlag.

[KKR08b] Michael Kuperberg, Klaus Krogmann, and Ralf Reussner. Performance Prediction for Black-Box Components using Reengineered Parametric Behaviour Models. In *Proceedings of the 11th International Symposium on Component Based Software Engineering (CBSE 2008), Karlsruhe, Germany, 14th-17th October 2008*, volume 5282 of *LNCS*, pages 48–63. Springer, Heidelberg, October 2008.

[KKR08c] Michael Kuperberg, Martin Krogmann, and Ralf Reussner. ByCounter: Portable Runtime Counting of Bytecode Instructions and Method Invocations. In *Proceedings of the 3rd International Workshop on Bytecode Semantics, Verification, Analysis and Transformation, Budapest, Hungary, 5th April 2008 (ETAPS 2008, 11th European Joint Conferences on Theory and Practice of Software)*, 2008.

[KKR09] Michael Kuperberg, Martin Krogmann, and Ralf Reussner. TimerMeter: Quantifying Accuracy of Software Times for System Analysis. 2009. To Appear at QEST '09.

[KLM+06] Tomás Kalibera, Jakub Lehotsky, David Majda, Branislav Repcek, Michal Tomcanyi, Antonin Tomecek, Petr Tuma, and Jaroslav Urban. Automated Benchmarking and Analysis Tool. In *Proceedings of the 1st International Conference on Performance Evaluation Methodolgies and Tools, VALUETOOLS 2006, Pisa, Italy, October 11-13, 2006*, volume 180 of *ACM International Conference Proceeding Series*, page 10. ACM, 2006.

[Kou05] Samuel Kounev. *Performance Engineering of Distributed Component-Based Systems - Benchmarking, Modeling and Performance Prediction*. Shaker Verlag, Ph.D. Thesis, Technische Universität Darmstadt, Germany, 2005.

[Kou06] Samuel Kounev. Performance Modeling and Evaluation of Distributed Component-Based Systems using Queueing Petri Nets. *IEEE Transactions on Software Engineering*, 32(7):486–502, July 2006.

[Koz08] Heiko Koziolek. *Parameter Dependencies for Reusable Performance Specifications of Software Components*. PhD thesis, University of Oldenburg, 2008.

[KR08] Klaus Krogmann and Ralf Reussner. *The Common Component Modeling Example*, volume 5153 of *LNCS*, chapter Palladio: Prediction of Performance Properties. Springer, Heidelberg, 2008.

[Kra08] Johan Kraft. Tracealyzer: The Lightweight Trace Visualizer for Embedded Systems, October 2008. http://www.tracealyzer.se/. Last visit: 2009-05-19.

[Kro07] Klaus Krogmann. Reengineering of Software Component Models to Enable Architectural Quality of Service Predictions. In Ralf Reussner, Clemens Szyperski, and Wolfgang Weck, editors, *Proceedings of the 12th International Workshop on Component Oriented Programming (WCOP 2007)*, volume 2007-13 of *Interne Berichte*, pages 23–29, Berlin, July31 2007. Universität Karlsruhe (TH).

[MAD94] Daniel A. Menascé, Virgilio A. F. Almeida, and Lawrence W. Dowdy. *Capacity planning and performance modeling: from mainframes to client-server systems*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1994.

[MAD04] Daniel A. Menascé, Virgilio A. F. Almeida, and Lawrence W. Dowdy. *Performance by Design*. Prentice Hall, 2004.

[Mah04] Qusay H. Mahmoud. Getting Started with Java Management Extensions (JMX): Developing Management and Monitoring Solutions. Online, January 2004. http://java.sun.com/developer/technicalArticles/J2SE/jmx.html. Last visit: 2009-05-19.

[MC07] Microsoft Corporation. Performance and Reliability Monitoring Step-by-Step Guide for Windows Server 2008. Online, April 2007. http://go.microsoft.com/fwlink/?LinkId=70270. Last visit: 2009-05-19.

[MC08] Microsoft Corporation. % Disk Time may exceed 100 percent in the Performance Monitor MMC. Online, February 2008. http://support.microsoft.com/kb/310067. Last visit: 2009-05-19.

[Mey07a] Marcus Meyerhöfer. *Messung und Verwaltung von Softwarekomponenten für die Performancevorhersage*. PhD thesis, Universität Erlangen-Nürnberg, Germany, 2007.

[Mey07b] Marcus Meyerhöfer. TestEJB: Response Time Measurement and Call Dependency Tracing for EJBs. In *MAI '07: Proceedings of the 1st workshop on Middleware-application interaction*, pages 55–60, New York, NY, USA, 2007. ACM.

[MM02] Adrian Mos and John Murphy. A Framework for Performance Monitoring, Modelling and Prediction of Component Oriented Distributed Systems. In *WOSP '02: Proceedings of the 3rd international workshop on Software and performance*, pages 235–236, New York, NY, USA, 2002. ACM.

[Mos04] Adrian Mos. *A Framework for Adaptive Monitoring and Performance Management of Component-Based Enterprise Applications*. PhD thesis, Dublin City University, Ireland, August 2004.

[Mur08] San Murugesan. Harnessing Green IT: Principles and Practices. *IT Professional*, vol. 10, no. 1:24–33, Jan./Feb. 2008.

[NPIF08] Yefim Natis, Massimo Pezzini, Kimihiko Iijima, and Raffaella Favata. Magic Quadrant for Enterprise Application Servers, 2Q08, April 2008. Gartner RAS Core Research Note G00156200. http://mediaproducts.gartner.com/reprints/oracle/article19/article19.html. Last visit: 2009-05-19.

[OCa] Oracle Corporation. Compatibility Statement for Oracle WebLogic Server 10g Release 3 (10.3). http://edocs.bea.com/wls/docs103/compatibility/compatibility.html. Last visit: 2009-05-19.

[OCb] Oracle Corporation. Configuring and Using the WebLogic Diagnostics Framework. Online. http://e-docs.bea.com/wls/docs103/wldf_configuring/. Last visit: 2009-05-19.

[OCc] Oracle Corporation. Understanding JIT Compilation and Optimizations. Online. http://download.oracle.com/docs/cd/E13188_01/jrockit/geninfo/diagnos/underst_jit.html. Last visit: 2009-05-19.

[OCd] Oracle Corporation. Using the WebLogic Persistent Store. Online. http://e-docs.bea.com/wls/docs103/config_wls/store.html. Last visit: 2009-05-19.

[OC08] Oracle Corporation. Oracle JRockit Mission Control Overview. Online, June 2008. White Paper. http://www.oracle.com/technology/products/jrockit/pdf/missioncontrol_whitepaper_june08.pdf. Last visit: 2009-05-19.

[Opd92] William F. Opdyke. *Refactoring Object-Oriented Frameworks*. PhD thesis, University of Illinois at Urbana-Champaign, 1992.

[pcm] Palladio Component Model. Online. http://www.palladio-approach.net/. Last visit: 2009-05-19.

[qim] Q-ImpreSS project site - quality impact prediction for evolving service-oriented software. Online. http://www.q-impress.eu/Q-ImPrESS/CMS/Overview/index_html. Last visit: 2009-05-19.

[qim08] Project Deliverable D2.1 Service Architecture Meta-Model (SAMM). Online, September 2008. http://www.q-impress.eu/Q-ImPrESS/CMS/Documents/D2.1-Service_architecture_meta-model.pdf. Last visit: 2009-05-19.

[RS08] Frank Rometsch and Horst Sauer. Dynatrace Diagnostics: Performance-Management und Fehlerdiagnose vereint. *iX*, 9/2008:72–75, 2008.

[RvHG+08] Matthias Rohr, André van Hoorn, Simon Giesecke, Jasminka Matevska, Wilhelm Hasselbring, and Sergej Alekseev. Trace-context sensitive performance profiling for enterprise software applications. In Samuel Kounev, Ian Gorton, and Kai Sachs, editors, *Performance Evaluation – Metrics, Models and Benchmarks: Proceedings of the SPEC International Performance Evaluation Workshop (SIPEW '08)*, volume 5119 of *Lecture Notes in Computer Science (LNCS)*, pages 283–302, Heidelberg, June 2008. Springer Verlag.

[RvHM+08] Matthias Rohr, André van Hoorn, Jasminka Matevska, Nils Sommer, Lena Stoever, Simon Giesecke, and Wilhelm Hasselbring. Kieker: Continuous Monitoring and on demand Visualization of Java Software Behavior. In *Proceedings of the IASTED International Conference on Software Engineering 2008*, pages 80–85. ACTA Press, February 2008.

[RvHM09] Matthias Rohr, André van Hoorn, and Nina Marwede. Tutorial for Kieker: Monitoring and Visualization of Software Behavior. Online, April 2009. Documentation. http://kieker.sourceforge.net/. Last visit: 2009-05-19.

[SBS06] Rima Patel Sriganesh, Gerald Brose, and Micah Silverman. *Mastering Enterprise JavaBeans 3.0*. John Wiley & Sons, Inc., New York, NY, USA, 2006.

[Sly06] Rebecca Sly. Introduction to the WebLogic Diagnostics Framework (WLDF). Online, June 2006. Dev2Arch Article. http://www.oracle.com/technology/pub/articles/dev2arch/2006/06/wldf.html. Last visit: 2009-05-19.

[Smi02] Connie U. Smith. *Encyclopedia of Software Engineering*, chapter Software Performance Engineering, pages 1545–1562. John Wiley & Sons, 2nd edition, January 2002.

[SPEC09] Standard Performance Evaluation Corporation. All SPECjAppServer2004 Results Published by SPEC. Online, 2009. http://www.spec.org/jAppServer2004/results/jAppServer2004.html. Last visit: 2009-05-19.

[Suc09] Akara Sucharitakul. Faban Driver Framwork, 2009. Project homepage. http://faban.sunsource.net/. Last visit: 2009-05-19.

[Sun99] Sun Microsystems, Inc. Simplified Guide to the Java 2 Platform, Enterprise Edition. Online, 1999. White Paper. http://java.sun.com/j2ee/reference/whitepapers/j2ee_guide.pdf. Last visit: 2009-05-19.

[SVBD04] Gunther Stuer, Kurt Vanmechelen, Jan Broeckhove, and Tom Dhaene. Sleeping in Java. In *Proceedings of the International Euromedia 2004 Conference, Hasselt, Belgium*, pages 74–78, April 2004.

[SW02] Connie U. Smith and Lloyd G. Williams. *Performance Solutions˜- A Practical Guide to Creating Responsive, Scalable Software*. Addison-Wesley, 2002.

[Sys00] Tarja Systä. *Static and Dynamic Reverse Engineering Techniques for Java Software Systems*. PhD thesis, University of Tampere, Finland, 2000.

[Szy02] Clemens Szyperski. *Component Software: Beyond Object-Oriented Programming*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2002.

[TMBS07] Paolo Tonella, Torchiano Marco, Bart Du Bois, and Tarja Systä. Empirical studies in reverse engineering: state of the art and future trends. *Empirical Softw. Engg.*, 12(5):551–571, 2007.

[WFP07] Murray Woodside, Greg Franks, and Dorina C. Petriu. The future of software performance engineering. In *FOSE '07: 2007 Future of Software Engineering*, pages 171–187, Washington, DC, USA, 2007. IEEE Computer Society.

[ZWL08] Tao Zheng, Murray Woodside, and Marin Litoiu. Performance Model Estimation and Tracking Using Optimal Filters. *IEEE Transactions on Software Engineering*, 34(3):391–406, 2008.