

Performance Management in Component-Oriented Systems Using a Model Driven Architecture™ Approach

Adrian Mos^{*}, John Murphy^{**}
Performance Engineering Laboratory
Dublin City University
{mosa,murphy}@eeng.dcu.ie

Abstract

Developers often lack the time or knowledge to profoundly understand the performance issues in large-scale component-oriented enterprise applications. This situation is further complicated by the fact that such applications are often built using a mix of in-house and Commercial-Off-The-Shelf (COTS) components.

This paper presents a methodology for understanding and predicting the performance of component-oriented distributed systems both during development and after they have been built. The methodology is based on three conceptually separate parts: monitoring, modelling and performance prediction. Performance predictions are based on UML models created dynamically by monitoring-and-analysing a live or under-development system. The system is monitored using non-intrusive methods and run-time data is collected. In addition, static data is obtained by analysing the deployment configuration of the target application. UML models enhanced with performance indicators are created based on both static and dynamic data, showing performance hot spots. To facilitate the understanding of the system, the generated models are traversable both horizontally at the same abstraction level between transactions, and vertically between different layers of abstraction using the concepts defined by the Model Driven Architecture. The system performance is predicted and performance-related issues are identified in different scenarios by generating workloads and simulating the performance models.

Work is under way to implement a framework for the presented methodology with the current focus on the Enterprise Java Beans technology.

1. Introduction

As companies continue to expose their business processes over the Internet for Business-to-Business (B2B) or Business-to-Consumer (B2C) interactions, the software systems they rely upon become increasingly

complex. The speed at which these software systems must be developed is also increasing due to the interest of each company to achieve a competitive advantage in their markets.

It can be argued that increasing the complexity and the time-to-market for software systems are two conflicting requirements. Other major conflicting requirements are ensuring that systems meet performance goals and reducing the costs at which these systems are developed.

Outsourcing parts of system development is a solution often used by enterprises to deal with development budget cuts. Even mission critical developments such as financial or military applications [1] increasingly need to resort to this approach. Another solution to the same problem is using Commercial-Off-The-Shelf (COTS) software. Both solutions may lead to situations where the developers responsible for the entire system do not profoundly understand the resulting software application. When the application is not clearly understood, it is often hard if not impossible to ensure that performance goals are met, especially if the system was not engineered for performance from the start.

Component oriented development [2] is gaining momentum mostly because it speeds up the development process for large enterprise systems. In addition, it forces developers to design with future changes in mind, which increases flexibility and reusability. A number of frameworks such as Sun's Enterprise Java Beans, OMG's Corba Component Model or Microsoft .NET are available and they can help reduce the development time and even help with performance and reliability issues such as scalability, fault-tolerance and availability. However, using such a framework does not guarantee performance, which is most often affected largely by the design of the application. Even though COTS components for these technologies are now available on component markets, we believe that the lack of Quality of Service (QoS) assurance is a major obstacle in their large-scale adoption.

We propose the Component Performance Assurance Solutions (COMPAS) framework that can help developers of distributed component-oriented applications realise and

^{*,**} The authors' work is funded by Enterprise Ireland Informatics Research Initiative 2001, and supported by Iona Technologies and Sun Microsystems Ireland.

correct performance issues during as well as after development. Another goal of the framework is deriving QoS parameters for COTS components, relevant to the application being built, so that different COTS components with the same functionality from different vendors can be compared.

We encourage the use of the presented framework together with Software Performance Engineering [3] methods continuously during the development process but major benefits can be obtained even when the proposed framework is used after the application has been developed.

The rest of the paper is organized as follows. In Section 2, an overview of COMPAS is presented. Section 3 describes the monitoring module together with a proof of concept implementation. The application modelling module and the Performance prediction functionality are presented in Section 4.

2. COMPAS Framework Overview

This paper describes a framework that can be used to detect and understand performance problems in distributed component-oriented applications based on technologies such as Enterprise Java Beans. In addition, the framework can help correct such problems, by providing means for comparison between different possible design solutions. We consider the following issues:

- Performance can be critical for large-scale component oriented applications.
- A poor architecture, a bad choice of COTS components or a combination of both can prevent achieving the application performance goals.
- Performance problems are more often caused by bad design rather than bad implementations.
- Often, performance is “a function of the frequency and nature of inter-component communication, in addition to the performance characteristics of the components themselves” [4].
- Fixing performance problems late in the development process is expensive.

To address these issues, COMPAS is structured into three main functional parts or modules that are interrelated:

- **Monitoring:** obtains real-time performance information from a running application without interfering with the application code or the application run-time infrastructure (i.e. the application server implementation). This paper is concerned with monitoring of EJB systems only.
- **Modelling:** creates UML models of the target application using information from the monitoring

module. The models are augmented with performance indicators and can be presented at different abstraction levels to improve the understanding of the application from a performance perspective.

- **Performance Prediction:** the generated models of the application are simulated with different workloads (e.g. corresponding to different business scenarios); simulation results can be used to identify design problems or poor performing COTS components.

There is a logical feedback loop connecting the monitoring and modelling modules. It refines the monitoring process by focusing the instrumentation on those parts of the system where the performance problems originate.

The intent of the presented framework is not to suggest a development process that prevents the occurrence of performance issues in the design, but rather to enable early discovery of such issues and suggest corrections.

2.1. Related Work

In the following paragraphs, we present related work and contrast our approaches in relation to it.

Monitoring

There is a significant amount of research and work in monitoring CORBA systems [5][6][7], however there are no existing generic EJB monitoring frameworks that can provide design level performance information (i.e. method and component level parameters). A number of application servers provide a certain degree of monitoring but most of them do so at a network/protocol level, giving little help to OO developers who want to understand which component/method is having the scalability problem. There are also a few third-party plug-ins [9][8] for application servers that would provide such monitoring information, however they are targeted at a specific application server, on a specific platform, offering little flexibility in choosing the development environment.

In a different category are EJB testing tools [10][11] that perform stress testing on EJB components and provide information on their behaviour. Such tools automatically create test clients for each EJB and run scripts with different numbers of simultaneous such clients to see how the EJBs perform. The main disadvantage of such a solution is the fact that it does not gather information from a real-life system but from separated components. Without monitoring the actual deployed system, it is difficult to obtain an accurate performance model for the entire system.

An interesting initiative in obtaining standardized performance data for EJB systems is the ECPerf [12] process. It defines a standard workload and standard

business applications that are generated during the testing process in order to determine the performance of application servers. Metrics like transaction throughput and response time are derived from the testing process and the results can be used by vendors to showcase their application server products. Although this approach does not involve monitoring of an arbitrary application, it is relevant to our research because it defines workload and metrics of interest to performance management of EJB systems.

Modelling and Prediction

The field of software performance modelling and prediction is vast. Important contributions have been presented in papers such as [13][14] reporting significant results in the improvement of the software development process, specifically the use of Software Performance Engineering methods aided by related tools such as SPE-ED. The techniques and the supporting tools require developers to create software and/or system models of the application under development. These models must have performance parameters such as I/O utilization, CPU cycles or network characteristics, specified by the developers in order for the performance predictions to generate meaningful results. It has been proved that such techniques and tools like SPE-ED help in achieving performance goals and reducing performance related risks for general object-oriented systems and even for distributed systems [14]. We argue that middleware such as EJB or other component-oriented platforms, exhibit an inherent complexity which developers find hard if not impossible to quantify even in simple models. Automated services such as caching, pooling, replication, clustering, persistence or Java Virtual Machine optimisations, provided by EJB application servers, for example, contribute to an improved and at the same time highly unpredictable run-time environment. Even more, application server implementation can vary greatly from vendor to vendor in respect to these services. Similarly, in CORBA (or CCM) based systems the search for performance improvements of the underlying framework under variable workloads leads to increased complexity [15]. It is therefore impossible for developers building such applications to create performance models where they specify the mapping of methods to processes or instances to processors, I/O characteristics or CPU utilisation. The framework presented in this paper extracts simplified performance data such as method execution time by monitoring live versions of the application under development, and creates UML performance models automatically. Such models can discover anti-patterns in the application implemented in a particular technology, which are not necessarily bad practices in other component technologies. The anti-pattern detection engine can have different profiles (e.g. one for EJB, one for

.NET) depending on the technology being used by the developers. A knowledge base such as [16] is used to drive the anti-pattern detection so that only relevant anti-patterns [17][18] are discovered for a particular technology. The generated UML models, like the SPE models, become increasingly detailed as more information is obtained, that is, as development progresses through iterations.

Predicting the performance of middleware-based systems has been approached in the past. Among the most common techniques are Petri-Nets [19] and Layered Queuing Network [13][21][20] models. We envisage that models created automatically by monitoring the system can be simulated and predictions derived for different workloads using LQN and Markov chains [19] techniques. Distinctive from related approaches, the proposed framework can perform the simulations at different abstraction levels, and developers can choose to refine particular simulations that are more likely to yield performance alerts. This can dramatically improve simulation duration and efficiency.

The main novelty of the COMPAS framework lies within the close integration between monitoring, modelling and performance prediction. Working with models at different levels of abstraction makes it easier to understand and simulate the target system. In addition, the focus on the performance of applications built with distributed component technologies such as EJB further differentiates our research from related work.

2.2. Key Features

Because model generation in the presented framework is dependent on monitoring information extracted from a running application, our approach integrates well within development environments that adhere to iterative development processes such as Rational Unified Process [22] or Extreme Programming [23]. Such processes demand that a running version of the application exists at the end of every iteration, making monitoring possible.

Models are represented in UML, which many enterprise-scale application developers are familiar with. The use of Model Driven Architecture (MDA) [24] and Enterprise Distributed Object Computing (EDOC) [25] concepts facilitates navigation between different layers of abstraction. The top-level models are represented using a technology independent profile, the Enterprise Collaboration Architecture (ECA) from EDOC, in order to benefit from a standardized form of representation for business modelling concepts. Lower level models are represented using UML specialized profiles such as the UML Profile for EJB [25] which provide means to illustrate technology specific details. Regardless of the

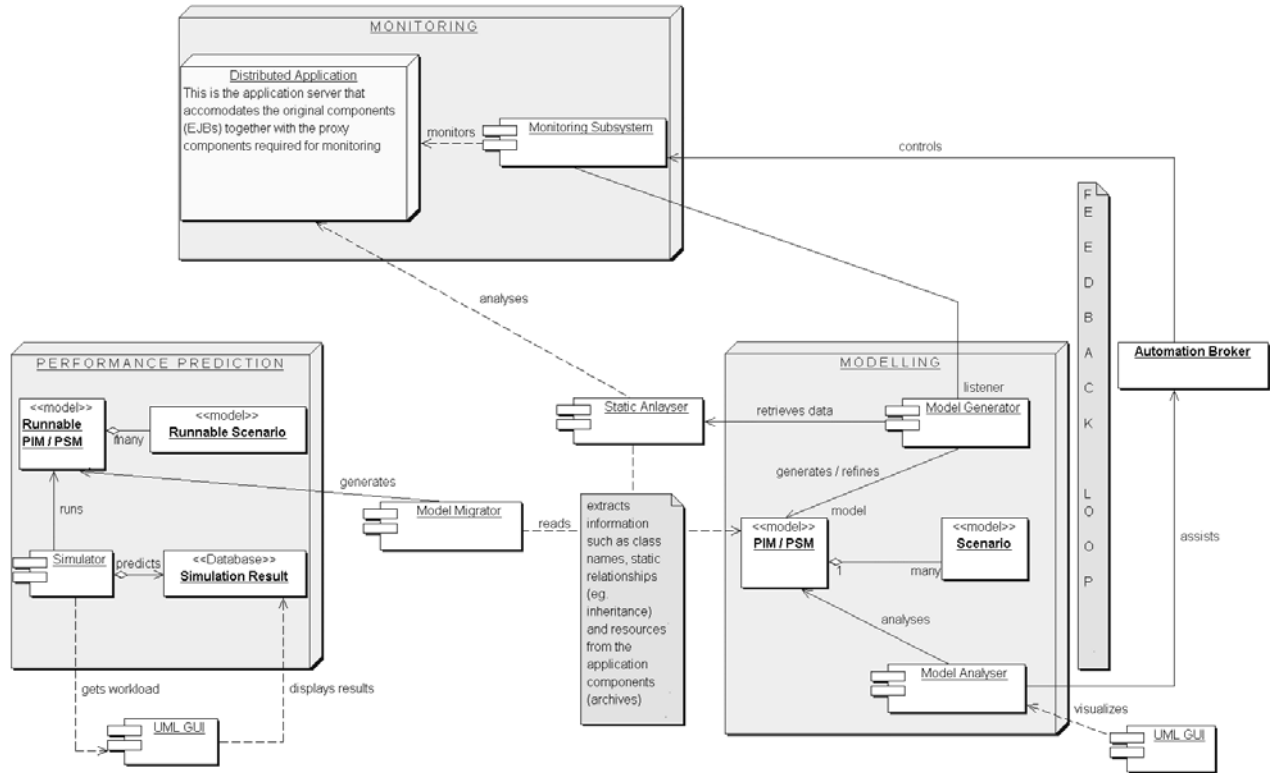


Figure 1. Framework Overview

level of abstraction, each model is augmented with performance information extracted by the monitoring module and presented using the UML Profile for Schedulability, Performance, and Time Specification [26].

The Performance Prediction Module uses runnable versions of the generated modules and simulates them with different workloads as inputs, displaying performance information in the same manner as in the modelling phase.

It is envisaged that both in the modelling phase as well as in the prediction phase, developers will navigate through the generated models in a top-down manner. If a performance alert is attached to a design element (during modelling or simulation), that element can be “zoomed into” and a lower-level more detailed model that includes that element is then inspected. This approach is highly intuitive, primarily because it is conceptually integrated with a typical design process in which high-level abstractions are found first, and then later refined into more-detailed abstractions, in an iterative manner.

A high level overview of the entire framework is pictured in Figure 1.

3. Monitoring

The performance management framework presented in this paper processes information extracted during the

monitoring process. Models are built using such information and in order for the models to be accurate, the monitoring module has to be reliable and efficient.

This section describes our approach to EJB applications monitoring and presents preliminary work. The monitoring module is currently focused on the EJB technology, however the rest of the framework is technology independent with the observation that technology specific profiles are used as inputs to performance prediction and pattern detection techniques. It is envisaged that the monitoring process described can be applied to any component-oriented platforms such as CCM or .NET.

3.1. General Considerations

By monitoring a system and extracting real run-time performance characteristics, the necessity to make assumptions on resource utilization or mapping of software entities to operating system or hardware resources, is greatly reduced. For complex middleware like EJB that handles tasks such as replication, concurrency, transactionality, pooling and caching in a variable manner (different from vendor to vendor or platform to platform), it becomes impossible to make such assumptions and monitoring provides a significant help.

In addition, for COTS components, it is likely that there is insufficient information to make any assumptions at all and in such situations, monitoring is critical.

However, run-time monitoring alone may not be sufficient to create meaningful models, because structural relationships such as inheritance and some associations are best extracted by static introspection of the target application. We use methods such as the Java Reflection API and XML parsing of application descriptors to extract static information from target applications.

By load stressing and monitoring an application using the presented framework, it is possible to derive performance metrics for existing COTS components, metrics that make sense from an enterprise point of view. Two important metrics for enterprise system are business transaction throughput and business transaction response time [12]. Different components from different vendors could then be compared for performance in the context of a particular application.

3.2. Enterprise Java Beans

Enterprise Java Beans [27] is a component architecture that allows the creation of scalable and reliable distributed applications. It is the core of the Java 2 Enterprise Edition (J2EE) platform and provides a number of complex services such as messaging, security, transactionality and persistence, which are provided by the container to any EJB component that requests them in the associated XML deployment descriptor.

Developers write code in compliance with the EJB specifications and deliver packaged components, consisting of the required classes [27] (component home interface, component remote interface which represents the component specification [2], bean class) and any other additional classes as required by the functionality of the component. Figure 2 illustrates such a component named *Developer Component*.

Application servers implement the home and remote interfaces, which are used by EJB clients. The home interface acts like a Factory [17] object and is responsible for creating/finding component instances. The EJBObject is an implementation of the remote interface and it acts as a wrapper for the original component instance. Its purpose is to facilitate the provision of additional services by the container (e.g. checking whether a particular client role has the credentials necessary for a method invocation)

The steps [27] a client is required to perform in order to access an EJB component are illustrated in Figure 2.

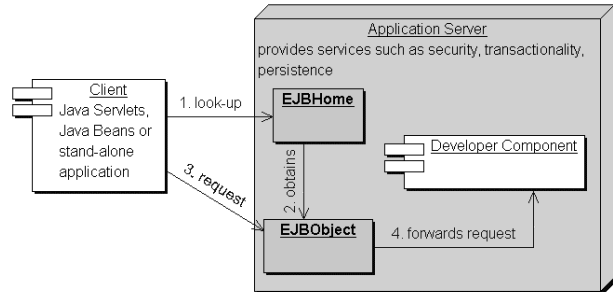


Figure 2. Client calling an EJB

3.3. Proxy Components

We have considered the following requirements when deciding on a monitoring approach:

- Portability: the framework must be used to instrument any J2EE compliant application running in any J2EE compliant application server
- Non-Intrusiveness: monitoring must not require changes to application source code or server implementation
- Low overhead: the overhead incurred by monitoring must not excessively affect the original system

To meet these requirements, the monitoring module uses proxy components to automatically generate a parallel application that mirrors the target application that must be monitored. For each component in the original application (target component), a Proxy [17] component that exposes the same remote interface [27] and uses the same Java Naming and Directory Interface (JNDI) [27] name, exists in the mirror application. The target components' original names registered in the JNDI directory, are changed upon the deployment of the mirror application. As illustrated in Figure 3, after the deployment of the monitoring infrastructure in the same run-time environment as the target application, clients will transparently communicate with the target components through the proxy components. This mechanism enables capturing of performance information by tracking all the invocations and EJB lifecycle events [27] for each monitored component.

The technology used by the monitoring module for managing the instrumentation of EJB components is Java Management Extensions (JMX) [28] which offers a lightweight, standardized way for managing Java objects. The inclusion of JMX in the J2EE standard assures that any J2EE compliant application server provides a JMX implementation.

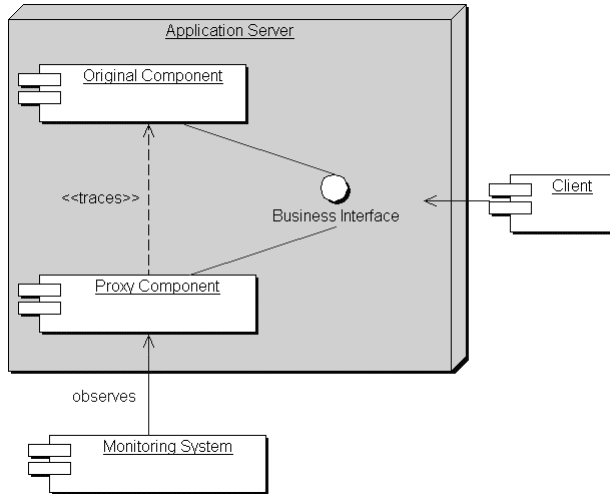


Figure 3. Using Proxy components

The JMX architecture has three levels:

- **Instrumentation level:** provides instant manageability to a manageable resource (any device, application or Java object) by using a corresponding MBean. A managed bean, or MBean for short, is a Java object that represents a JMX manageable resource. MBeans follow the JavaBeans components model, thus providing a direct mapping between JavaBeans components and manageability. Because MBeans provide instrumentation of managed resources in a standardized way, they can be plugged into any JMX agent.
- **Agent level:** provides management agents. JMX agents are containers that provide core management services which can be dynamically extended by adding JMX resources. A JMX Agent is composed of an MBean server, a set of MBeans representing managed resources, and at least one protocol adaptor or connector. Protocol adaptors create a representation of the MBeans into another protocol, such as HTML or SNMP. Connectors include a remote component that provides end-to-end communications with the agent over a variety of protocols (for example HTTP, HTTPS, IIOP).
- **Manager level:** provides management components that can operate as a manager or agent for distribution and consolidation of management services. A JMX manager provides an interface for management applications to interact with the agent, distribute or consolidate management information, and provide security. JMX managers can control any number of agents, thereby simplifying highly distributed and complex management structures.

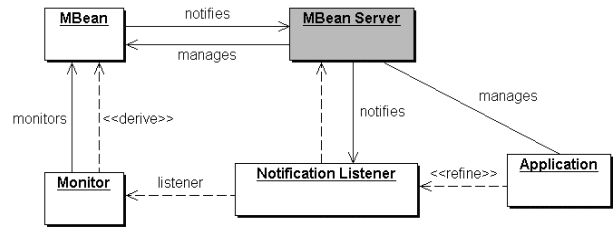


Figure 4. Java Management Extensions

Figure 4 shows that the MBeans are managed by an application through the MBean Server. In addition, they can be monitored by a special type of MBeans, Monitors that can observe changes in the state of a monitored MBean and notify the registered listeners. An MBean corresponds to a managed resource and it can interact with that particular resource. In our framework, MBeans correspond to Proxy Components that are responsible for tracking events in the target application. In Figure 5, the integration of JMX into the monitoring framework is illustrated. Proxy Component beans extend the MBean abstraction but they are depicted as separate entities in the diagram for presentation purposes.

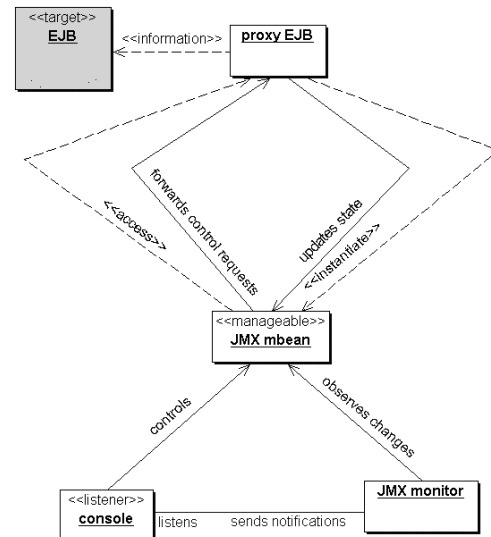


Figure 5. Monitoring EJBs with JMX

One of the main reasons for choosing JMX as the infrastructure technology for the monitoring module is that it provides a flexible way to control the proxy components. Monitoring can be activated or deactivated for any particular component. The activation/deactivation decisions are taken by an independent automation broker, which uses information generated by the modelling module about the structure of the application (see Figure 1). Another major advantage of using JMX is that it is a very lightweight technology, helping reduce the overhead of the monitoring infrastructure.

The proxy components are automatically generated by inspecting the original application in its pre-deployment



Figure 6. Monitoring Consoles

form (i.e. an enterprise application archive [27] file .EAR). For each component in the original application, a copy is generated representing its proxy counterpart. The copy is modified by replacing the bean class (component implementation [2]) with a new implementation acting as a Proxy [17] for the original component. The rest of the component is kept identical to the original component. The new set of components comprising the proxy application is packaged and deployed onto the same application server as the target application.

A proof of concept monitoring module has been implemented. It is capable of intercepting method calls and basic EJB life-cycle events such as creation and destruction of instances. In addition, it can generate time-stamps, which are useful in determining the dynamic structure of the system. Basic graphical consoles are used to display such information in real-time (see Figure 6).

In previous work [29] we have described how the monitoring module can be used to detect performance issues even when used separated from the complete

framework. Experimental results demonstrating the low-overhead and high-accuracy characteristics have been presented and basic modelling consideration have been described [29]. Although the results presented in [29] relate to a previous implementation of the monitoring infrastructure using the Java Messaging Service API, recent results, based on a JMX implementation, confirm its low overhead and high-accuracy characteristics.

4. Performance Modelling and Prediction

The main goal of the presented framework is to help developers of large enterprise component-oriented applications find and predict performance problems in their systems, using concepts and visual representations that they easily understand.

Based on information extracted by the monitoring module, UML models are generated which show where performance problems are located. By simulating such models, predictions are made that help understand the

implications of changes in workload or changes in QoS characteristics for particular components. Having such prediction data, developers can make informed design decisions and choose the best COTS components to meet the application needs. Models are also used to increase the efficiency of the monitoring process by activating monitoring only for those components that are responsible for performance problems, and deactivate the monitoring of the other components. This activation/deactivation process is continuous and as it is envisaged that as models are being refined, the monitoring overhead decreases.

The next two sub-sections briefly present the Model Driven Architecture and the performance ontology that the framework uses. The remainder of the section describes the modelling and prediction functionality of the framework.

4.1. Model Driven Architecture (MDA)

The Model Driven Architecture [24] proposes a new approach to the development and understanding of complex systems and promotes portability across the main platforms that are currently in use now or will be used in the future.

MDA introduces two important concepts, the Platform Independent Model (PIM) and the Platform Specific Model (PSM). A PIM would generally be used in the earlier stages of development and it consists of a detailed UML model of the business logic without any technological details. For example, at the beginning of a development process, developers would model business entities such as <Account>, <Bank> and their behaviour which are all completely platform independent there is no need for any platform specific information, such as EJB Home Objects. Note however that a platform can be anything from a hardware platform, to operating system to middleware to another PIM. So the notion of platform and platform independence are relative, which makes it possible to have an arbitrary number of PIMs for the same problem space, each representing a different level of abstraction. A PSM has platform specific information in the model, such as EJB or CORBA stubs. Again, taking into consideration the relative aspect of a platform, a PSM can be just a more detailed description of a PIM, with more technical details.

A major advantage of using MDA is that models at different levels of abstraction can be implemented for different platforms, that is, from a set of PIMs, a large combination of PSMs could be realized, and the entire application would preserve its integrity. For example for a business application, for the same set of PIMs (the suite of models that describe the system at a platform independent level), different combinations of PSMs could be derived for each PIM. An internal banking PIM could be realized

by using an EJB mapping [25] to generate EJB PSMs. The B2B PIMs could be realized by using XML and SOAP PSMs. All these PSMs would interoperate with each other as specified in the PIMs. If for some reason, there is a need to generate the B2B PSMs in CORBA, that would not affect any other models and the generated system would be cohesive.

MDA facilitates “zooming in” and “zooming out” at different abstraction/realization levels. A PIM can be “zoomed into” to browse the PSMs that realize it, or a PSM could be “zoomed out of” to inspect the upper layer of abstraction. This facility is central to the presented performance management framework because it enables navigation between different refinement application layers when increased precision is needed for pinpointing a performance issue presented at the top levels of the application models hierarchy.

A simple illustration of MDA concepts is provided by Figure 7.

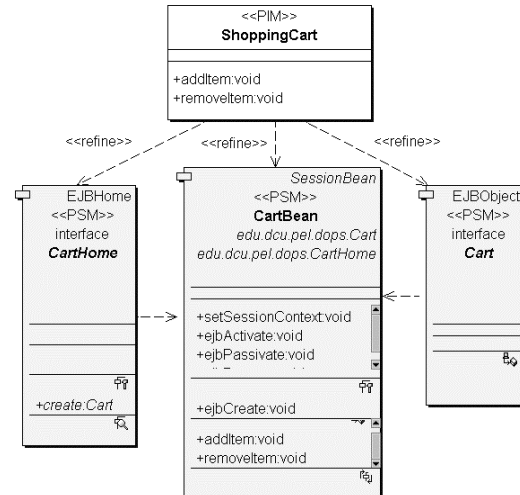


Figure 7. Mapping a simple PIM to an EJB PSM

4.2. Performance Modelling Ontology

The UML Profile for Schedulability, Performance, and Time Specification [26] defines the ontology used for performance models in the presented framework. Some of the main concepts in the ontology are:

Performance context: “specifies one or more scenarios that are used to explore various dynamic situations involving a specific set of resources.”[26]

Scenario: “...a sequence of one or more scenario steps. The steps are ordered and conform to a general precedence/successor relationship. Note that, in the general case, a scenario may involve multiple threads due to forking within the scenario.”[26]

Step: “An increment in the execution of a particular scenario that may use resources to perform its function. In

general, a step takes finite time to execute. It is related to other steps in predecessor/successor relationships.”[26]

Resource: “An abstraction view of passive or active resource, which participates in one or more scenarios of the performance context.”[26]

To simplify the presentation of performance models and increase visibility of generated sequence diagrams, *anonymous timing marks* [26] are used.

4.3. Performance Management Functionality

Performance models are generated at run-time based on measurements taken by the monitoring module. Two major sets of data are obtained during the monitoring process:

- Model generation data: component instances [2][27] are monitored for method invocations and lifecycle events. Time-stamps (such as those provided by JMX) are used together with component instance IDs, method names and method execution times to order events and build statistical dynamic models of the running application.
- Performance information: metrics such as response times and throughput are determined for the runtime entities and are used to augment the generated UML models.

When using model generation data to detect models in the monitored application, techniques such as Markov chains, Petri Nets and queuing networks are used. Statistical results based on large number of measurements are used to determine scenarios in the system, starting at previously determined points of entry. For example, in an EJB system, such a point of entry could be a web layer component such as a product selection list in a retail application. Such a determined scenario could be one corresponding to a “buying an item” use-case. Another could correspond to a “write a product review” use-case.

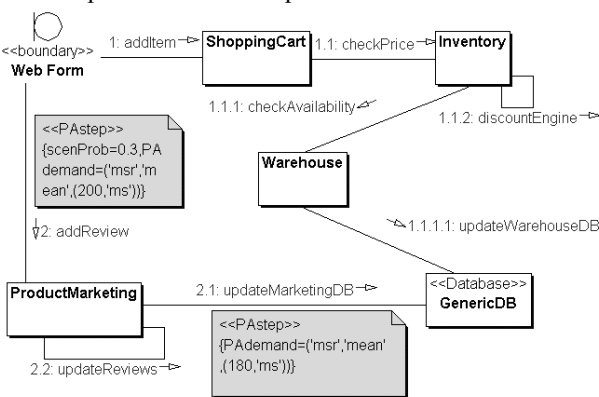


Figure 8. Scenarios with probability and performance parameters

Models representing these scenarios would have performance related information in addition to probabilities.

Figure 8 illustrates this example. The first scenario starts with step “1. addItem” and the second scenario with step “2. addReview”. Please note that these scenarios do not illustrate a real design, but rather a very simplistic imaginary example.

To reduce visual cluttering, there are only two annotations regarding performance and probabilities in the example diagram, however we envisage a framework implementation that features an efficient way of dealing with such visual elements by selectively hiding or showing elements depending on user preferences. Scenario 2 (represented by the first step in the scenario) has a probability of occurrence of 30% and a mean execution time of 200ms. One of the steps in scenario 2, step “2.1 updateMarketingDB” has an associated mean execution time of 180ms, representing 90% of the total scenario execution time. Even though the example diagram is a UML collaboration diagrams, models can be presented using sequence and activity diagrams as well. To improve understanding of such diagrams, Object Constraint Language [30] (OCL) notations may be used together with statistic data to explain the conditions, in which a particular scenario occurs, not just the probability of occurrence. For example it can be statistically determined that a scenario is followed only when a parameter passed to the top-level scenario component, has a particular value.

Models such as the one presented in Figure 8 are generated during the monitoring process or by a later analysis of the monitoring logs. They are augmented with performance attributes such as “mean response time”. Based on user-defined rules, performance alerts are issued by the modelling environment, when certain conditions such as “too much growth in execution time” or “scenario throughput > user defined value” are met. If the user defines values such as expected mean and maximum values for a particular scenario response time, the models will show alerts in those areas exceeding these values. If the user does not specify such values, the framework can still suggest possible performance problems when certain conditions like “the response time increases dramatically when small numbers of simultaneous scenario instances are executed” are encountered. If a particular step in the affected scenario is mainly responsible for the degradation of scenario performance parameters, that step is identified and the alert narrowed down to it. Figure 9 and Figure 10 illustrate how a performance problem can be narrowed down using the MDA approach. Both diagrams are PIMs, however, it can be imagined that developers can proceed to lower levels such as EJB PSMs to identify technology

specific events such as lifecycle events that can cause performance degradation.

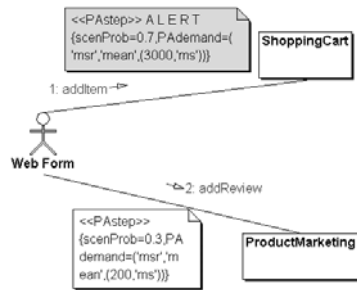


Figure 9. Top level PIM showing a performance alert

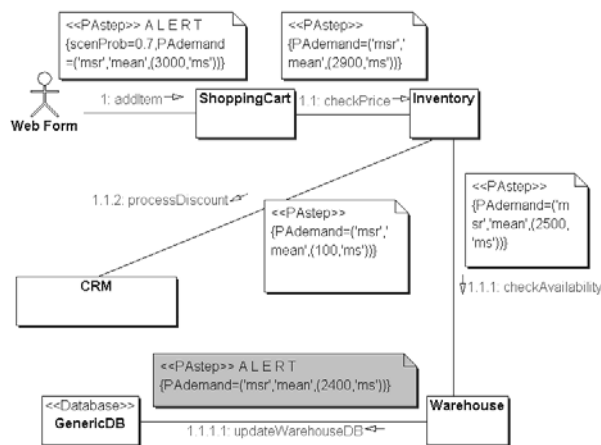


Figure 10. Identifying performance degrading steps

When browsing the generated models using the MDA approach, the top level abstractions are usually represented by the first steps in particular scenarios. A top-level model representing a scenario can represent just the first step of the scenario with the performance attributes such as response time or throughput associated. As developers navigate down the system tree, more scenarios/steps are revealed.

Predicting performance involves simulating the generated models. The users can specify workload characteristics [26] such as the number of simultaneous users and their inter-arrival rate. Expected performance attributes can also be specified. Workloads are then used to simulate the models. Users can easily change workload characteristics and re-run the simulation. The same mechanisms for generating performance alerts are used in the simulation stage, as in the monitoring/modelling stage. Developers can even modify the generated models and observe the effects the changes have on the overall performance, by simulating the altered models.

In both the monitoring/modelling stage and prediction stage, models can detect bad design practices. For

example, an EJB PSM can show a performance alert when an entity bean [27] finder method returns a large result set. In such a situation, a pattern [17] such as Value List Handler [16] is suggested by the framework to alleviate the performance problem.

5. Conclusion

The increasing complexity of large-scale enterprise applications, fast time-to-market cycles, and budget cuts are driving corporations to adopt measures such as outsourcing and adoption of COTS components. Technologies such as Enterprise Java Beans or .NET are used in order to ease and speed-up development. Under these circumstances, performance requirements are difficult to fulfil, in particular for development teams that postpone dealing with performance issues in their design until it becomes too late and too expensive to correct.

We propose a framework that addresses these issues by automating the discovery of performance problems. It integrates three functional modules: monitoring, modelling and performance prediction. The monitoring module is responsible for obtaining information from a running component-oriented application. The information is used by the modelling module to generate models of the application that have performance information associated to each entity. Models can be traversed in an MDA manner at different levels of abstraction. The performance prediction module uses the generated models to perform simulations that can be used to predict the performance of the system when workload or design changes. The framework is best used in iterative development environments that use processes such as RUP, because in such environments a running version of the system exists after each iteration.

Work to implement the framework is in progress. We have implemented a proof of concept non-intrusive monitoring module for EJB systems, using JMX as the instrumentation support. It extracts useful data such as method invocation performance attributes and lifecycle events such as component instantiation. We are considering methodologies such as Markov chains, Queuing Networks and Petri Nets for building models. Model Simulators and extensible UML visual environments such as the open-source ArgoUML [31], are being evaluated for integration in the framework.

6. References

- [1] C. Albert, L. Brownsword, "Meeting the Challenges of Commercial-Off-The-Shelf (COTS) Products: The Information Technology Solutions Evolution Process (ITSEP)", *International Conference on Component Based Software*

- Systems (ICCBSS) 2002, LNCS 2255*, Springer-Verlag Berlin Heidelberg, 2002, pp. 10-20
- [2] J. Cheesman, J. Daniels, *UML Components*, Addison Wesley, October 2001
- [3] C. U. Smith, *Performance Engineering of Software Systems*, Addison-Wesley, 1990
- [4] P.C. Clements, "Coming Attractions in Software Architecture", *No.CMU/SEI-96-TR-003*, Software Engineering Institute, Carnegie Mellon University, February 1996
- [5] F. Lange, R. Kroeger, M. Gergeleit, "JEWEL: Design and Measurement of a Distributed Measurement System", *IEEE Transactions on Parallel and Distributed Systems*, November 1992
- [6] B. Sridharan, S. Mundkur, A.P. Mathur, "Non-intrusive Testing, Monitoring and Control of Distributed CORBA Objects", *TOOLS Europe 2000*, St. Malo, France, June 2000
- [7] R. Weinreich, W. Kurschl, "Dynamic Analysis of Distributed Object-Oriented Applications", *Proc. Hawaii International Conference On System Sciences*, Kona, Hawaii, January 6-9, 1997
- [8] Precise, *Precise/Indepth for the J2EE platform*, www.precise.com/products/indepth.asp
- [9] Sitraka, *PerformaSure*, <http://www.sitraka.com/software/performasure/>
- [10] Empirix, *Bean Test*, www.empirix.com/empirix/web+test+monitoring/products/
- [11] Segue, *SilkTest*, www.segue.com/html/s_solutions/s_silktest/s_silktest_toc.htm
- [12] Sun Microsystems Inc., *ECperf Specification 1.1 Public Review Draft*, February 19, 2002
- [13] L.G. Williams, C.U. Smith, "Performance Engineering Evaluation of Software Architectures", *Proc. First International Workshop on Software and Performance (WOSP'98)*, Santa Fe, NM, USA, October 1998
- [14] C.U. Smith, L.G. Williams, "Performance and Scalability of Distributed Software Architectures: An SPE Approach", *Parallel and Distributed Computing Practices*, 2002
- [15] I. Abdul-Fatah, S. Majumdar, "Performance of CORBA-Based Client-Server Architectures", *IEEE Transactions on Parallel and Distributed Systems*, Vol. 13, No. 2, February 2002
- [16] J. Crupi, D. Alur, D. Malks, *Core J2EE Patterns*, Prentice Hall, 30 September, 2001
- [17] E. Gamma, R. Helm, R. Johnson, J. Vlissides, *Design Patterns – Elements of Reusable Object Oriented Software*, Addison-Wesley, NY, 1995.
- [18] A. I. Verkamo, J. Gustafsson, L. Nenonen, J. Paakki, "Design Patterns in Performance Prediction", *Proc. 2nd International Workshop on Software and Performance (WOSP'00)*, Ottawa, Canada, September 2000
- [19] E. Dimitrov, A. Schmietendorf, R. Dumke "UML-Based Performance Engineering Possibilities and Techniques", *IEEE Software*, Vol. 19, No. 1, January/February 2002
- [20] T.-K. Liu, S. Kumaran, Z. Luo, "Layered Queuing Models for Enterprise JavaBean Applications", *Proc. 5th International Enterprise Distributed Object Computing Conference (EDOC)*, Seattle, WA, USA, 4-7 September 2001
- [21] D.C. Petriu, H. Amer, S. Majumdar, I. Abdul-Fatah, "Using Analytic Models for Predicting Middleware performance", *Proc. 2nd ACM Int. Workshop on Software and Performance (WOSP'00)*, Ottawa, Canada, September 2000
- [22] P. Krutchen, *Rational Unified Process*, Addison Wesley, April 2000
- [23] K. Beck, *EXtreme Programming EXplained*, Addison Wesley, 31 October, 1999
- [24] Object Management Group, *Model Driven Architecture*, OMG document number ormsc/2001-07-01, OMG, 2001
- [25] Object Management Group, *UML Profile for Enterprise Distributed Object Computing Specification*, OMG document number ptc/02-02-05, OMG, 2002
- [26] Object Management Group, *UML Profile for Schedulability, Performance, and Time Specification*, OMG document number ptc/02-03-02, OMG, 2002
- [27] E. Roman, S. W. Ambler, T. Jewell, *Mastering Enterprise Java Beans Second Edition*, Wiley Computer Publishing, 2002
- [28] M. Fleury, *JMX: Managing J2EE with Java Management Extensions*, Sams, 8 February 2002
- [29] A. Mos, J. Murphy, "Performance Monitoring of Java Component-Oriented Distributed Applications", *Proc. IEEE 9th International Conference on Software, Telecommunications and Computer Networks (SoftCOM)*, Croatia/Italy, October 2001
- [30] J. Rumbaugh, I. Jacobson, G. Booch, *The Unified Modelling Language Reference Manual*, Addison-Wesley 1999, pp 367.
- [31] Tigris.org, *ArgoUML*, <http://argouml.tigris.org>