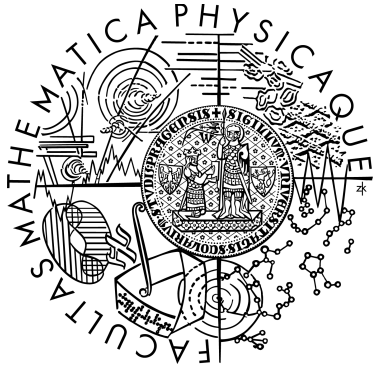
Charles University in Prague
Faculty of Mathematics and Physics

Doctoral Thesis



Lubomír Bulej

Connector-based Performance Data Collection for Component Applications

Department of Software Engineering
Advisor: Doc. Ing. Petr Tůma, Dr.

Annotations

Title

Connector-based Performance Data Collection for Component Applications

Author

Ing. Lubomír Bulej
e-mail: *lubomir.bulej@dsrg.mff.cuni.cz*, phone: +420 221 914 267

Department

Distributed Systems Research Group, *http://dsrg.mff.cuni.cz*
Department of Software Engineering, Faculty of Mathematics and Physics
Charles University in Prague, Czech Republic

Advisor

Doc. Ing. Petr Tůma, Dr.
e-mail: *petr.tuma@dsrg.mff.cuni.cz*, phone: +420 221 914 267

Mailing address

Charles University in Prague, Dept. of SW Engineering,
Malostranské nám. 25, 118 00 Prague, Czech Republic

Abstract

In this work, we propose a generic approach to collection of performance data for heterogeneous component-based applications with the aim to provide easier and less costly access to performance data needed for measurement and model-based performance analysis of component applications.

The technical foundation for the approach is built on generic solutions to various aspects of performance data collection and is made of three parts. The first part provides a design of a generic measurement infrastructure which handles common performance measurement tasks and allows collecting arbitrary performance data in response to performance events. The second part proposes using architecture-based connectors for instrumentation of component applications and provides a design of a performance instrumentation connector element for use with the measurement infrastructure. The third part proposes integration of connectors into deployment process of component applications which enables deployment and transparent instrumentation of heterogeneous component-based applications.

Keywords

performance measurement, connector-based instrumentation, heterogeneous component-based applications

Acknowledgments

This work builds on the results and experience arising from several years of research under the umbrella of the Distributed Systems Research Group at Charles University, Prague. I am deeply grateful to Petr Tůma, the supervisor of my doctoral studies at Charles University, for introducing me to the world of academic research and for his support, both personal and professional, during the past years. I am also grateful to František Plášil, the head of the Distributed Systems Research Group, for creating an inspiring research environment, and for his valuable comments and professional support.

My colleagues Tomáš Bureš, and Tomáš Kalibera, and my advisor Petr Tůma deserve a special acknowledgment for long-term collaboration on the research which provided the motivation for this work and led to some of the solutions presented in this thesis. I would like to remember Adam Buble, for his friendship, and for his invaluable contribution to our research on middleware performance evaluation.

I would also like to thank other members of the Distributed Systems Research Group, for making the research group a success, for sharing the highs and lows of doctoral studies, and for various kinds of support they have kindly provided. In particular, my thanks go to Jiří Adámek, Martin Děcký, Petr Hnětynka, Viliam Holub, Pavel Ježek, Jan Kofroň, Vladimír Mencl, Pavel Parízek, Tomáš Poch, and Ondřej Šerý.

Martin Děcký deserves a special acknowledgment for taking over my teaching duties during the final stage of writing this thesis.

This work and the related research was partially supported by the Grant Agency of the Czech Republic projects 102/03/0672, 201/03/0911, 201/05/H014, and 201/06/0770, by the Czech Academy of Sciences project 1ET400300504, by the Ministry of Education of the Czech Republic project MSM0021620838, and the ITEA/EUREKA projects OSMOSE and OSIRIS.

And last, but not least, I am indebted to my parents and family for their support and encouragement during my doctoral studies. In particular, I am indebted for life to my wife Kristýna, for her patience, support, and single-handed care of our son Lubomír and our yet unborn daughter. All of them had to endure my absence in family life so I could finish this work.

Table of Contents

List of Figures

Chapter 1

Introduction

The performance of a computer system is primarily determined by the performance of its hardware and software components. In the past decades, the performance of the hardware parts, and specifically of processors, has been following the trend predicted by Moore's law, despite the increasing complexity of hardware designs. In the case of software parts, the complexity has been also steadily increasing, but the perceived performance does not appear to change much.

With the processor performance increasing, the software performance has to be degrading to maintain the perceived unchanging level of performance. In fact the increase in processor performance has been primarily driven by demand for faster computers to allow existing software perform better, which in turn enabled development of more complex software, which again required faster processors to provide better performance.

In contrast to processor market though, the software market is mainly driven by demand for features and functions, with performance trailing far behind. Consequently, most software projects are little concerned about performance, with the exception of few specific domains, such as high performance computing, gaming software, and software for real time and mission critical systems.

However, performance has a different meaning in high performance computing and in real time systems. While in the former performance is related mainly to efficiency of computation with focus on performing maximum amount of work per unit of processor time, in the latter performance is related mainly to satisfying timing constraints with focus on organizing and scheduling the work that needs to be performed to meet various hard and soft deadlines. In both cases though, the performance must be measured, tracked, and analyzed to obtain information that can be used for steering the development process.

Traditional software development processes are mainly focused on achieving the goals in terms of features and stability while avoiding defects that are costly to repair. Functional correctness is tested during development by regression tests which exercise isolated functions of the software project, and overall robustness is tested against various stress scenarios. Systematic evaluation and analysis of performance is rarely part of the quality assurance process.

With the increased use of software as a means to provide complex, large-scale services, the performance of the system becomes an important aspect of service quality and is a key aspect of user experience and perception of the service. While this is also true for end-user applications targeted at specific tasks, or even middleware platforms, the performance of large-scale services is more visible to a user and requires increased attention [Hof05, Rog05] to maintain customer satisfaction.

Moreover, providing sufficient service performance for large scale services by means of ever-powerful hardware becomes more costly, as the increase in hardware performance is far below the increase in power consumption. For this reason, providers of large scale

services may focus on different hardware architectures to keep the power costs under control [Bar05, Lau05]. Under the same power envelope, properly written applications may achieve better throughput on multi-core processors and in distributed environment, but the architecture of such applications is more difficult to design. Also, with less computing power available to a single execution context, the implementation of such applications should strive for maximum efficiency.

Consequently, a modern software development process should, in addition to functional testing, also periodically collect performance data for the software project and include performance tracking and analysis in the arsenal of methods used in the quality assurance process.

Since preparing an infrastructure for collecting performance data for a software project is an arduous task, there should be a common infrastructure for performance data collection. Unfortunately, given the diversity of software projects, it is difficult to create an infrastructure that would suit all or most purposes. This coupled with lack of interest to create a generic performance evaluation platform results in most work related to collection of performance data and performance evaluation to be repeated over and over.

We aim to contribute to the solution of this problem and in this thesis we propose a generic approach to performance data collection for component applications. The approach exploits the specific characteristics of component-based software development, such as component composition and explicit description of application architecture.

The approach is supported by design of a generic measurement infrastructure that can be used for collecting performance data and an instrumentation method suitable for component applications. A long term goal is to provide a common infrastructure that could be used for collection of performance data needed for performance analysis.

To better prepare context for elaborating the motivation and the goals of this thesis, we first provide a brief overview of software performance analysis techniques that are commonly used to track, analyze, and predict performance of software systems, and point out the role of performance data collection in their application.

1.1 Software Performance Analysis

Software performance analysis research has two main branches. One branch is primarily theoretical, based on describing and analyzing system models created using various methods, while the other is mainly empirical, based on observation and measurement performed on real or prototype systems. In the following text, we will use the term system under study to denote the subject of performance analysis, and the term system under test to denote a system or model, which is subjected to application of performance analysis methods in order to derive the quantitative properties of the system under study.

The distinction between system under test and system under study is intentional. Very often they will be identical, however in many cases, especially when the complexity of the system under study limits the feasibility of performance analysis, the system under test will be represented by a model or another approximation of the system under study.

The following overview presents the two main branches of performance analysis research along with their typical applications. The two branches differ mainly in the methods used for performance analysis and also into the character of a system under test. The theoretical branch is based on creating and analyzing models, therefore the system under test is always a model of the system under study. The empirical branch is based on observation of the system under test, therefore the system under test may be identical to

the system under study, it if exists. The application of methods from either branch does not exclude methods from the other branch. In fact, they can be often used in complementary ways.

1.1.1 Model-based analysis

Performance modeling is a discipline in which the system under study and the system under test are distinct. The system under test is a model of the system under study, which captures primary characteristics of the original system relevant to its performance, while secondary characteristics of the original system are discarded or approximated. One of the reasons for creating performance models is the already mentioned complexity of a system under study. Another reason is that in some situations, such as when designing system architecture, the system under study may not yet exist.

For performance analysis, what a system really does is not important, but when and how long it takes. The behavior of a system in time can be thus reduced to occurrence of events triggering execution of actions with non-zero duration time. A performance model must therefore allow capturing the conditions leading to execution of different actions or otherwise describe how often, when, and in what context they are executed. This is typically achieved by creating an analytic models based on formal description of the system behavior and the actions it executes, or by creating a simulation model, which mimics the behavior of a system under study.

Simulation models describe a system with a high level of detail and provide very accurate results. However, their major drawback is typically long simulation time needed to obtain results. Analytic models, on the other hand, abstract away from low-level details and typically resort to stochastic description of system properties. While analytic models provide less accurate results, they typically require several orders of magnitude less time to obtain results from a model. Hybrid approaches that combine simulation and analytical models can be found as well.

Formal methods suitable for performance analysis have their roots in traditional formal methods devised for modeling and functional analysis of concurrent systems, such as *process algebras* and *petri nets*. However, the traditional methods abstract away from time needed to complete individual actions and are only interested in relative order of those actions, which makes them infeasible for performance analysis where time needs to be quantified. Therefore many of the traditional formal methods were modified to support the notion of non-unit time associated with actions performed by a system, resulting in the genesis of formal methods such as *stochastic process algebras* [GH+95, Hil96, BG98], *stochastic petri nets*, etc.

Other methods such as *queuing networks* are based on queuing theory in which time and probability of event occurrence were the key parameters since its invention. Since the concepts from queuing theory can be easily mapped to concepts such as process, resource, processing time, etc., the use of queuing networks has become a natural choice for modeling performance of computer systems. The concept of *layered queuing networks* [AB83, Woo89, RS95] evolved from applying queuing networks to modeling of performance of distributed systems and is better suited for modeling contention of software processes for shared resources.

Using a model to determine the value of a performance attribute concerning a system under study (such as the rate at which a system processes requests depending on the number of arriving requests) requires several steps. First, the attributes from the domain of a system under study have to be mapped to the model domain, which requires

determining which model properties correspond to the performance attributes of the system under study. Then the model must be solved and the results must be mapped back to the domain of the system under study.

Depending on the type of model, the solution can be found either analytically or by simulation. In many cases, analytic solution can be found only for a limited class of models. Simulation, on the other hand, is applicable to a substantially wider range of models. However, because of their long running times, it is advantageous to use compositional methods that provide means to compose a model from smaller and simpler models. This reduces the internal complexity of a model and makes modeling complex systems feasible, both in terms of managing and solving a model.

Since performance modeling is like any other modeling activity, the quality of the results reflects the quality of a model. Besides structural information, a model also contains many parameters which encode the stochastic properties of a system. Creating a good performance model therefore requires experience and good understanding of a system under study to correctly set various model parameters. The values of model parameters can be also obtained from performance measurements carried out on the system under study or its prototype.

Interestingly, while composition helps reducing model complexity, it can make the model more difficult to calibrate, because composition may shift the semantics of various model parameters. This is an issue especially when composing component performance models with parameters calibrated using performance measurement techniques, because typically the calibration is performed under ideal conditions when a component does not share resources with other components. There is little research on how resource sharing in presence of other components influences the performance of a component and thus the original model parameters.

With the advent of model driven software development in recent years, the focus of performance modeling research shifted on deriving performance models from software design models. The motivation is to use the structural and behavioral information contained in those models to avoid creating a standalone performance model duplicating a lot of information already contained in the design model. This approach also has the potential to bring performance modeling closer to software engineers, alleviating to a certain degree the need for performance experts creating separate performance models [HWR99, LFG04, LG05].

Since the most widespread modeling language is UML, there are now methods for transformation of UML design models to queuing networks [CM00, SW02], layered queuing networks [SG98, GP02, PS02, GP05], stochastic petri nets [KP00, BDM02, LMC02, GH+04], stochastic process algebras [CG+04, GK05], and simulation models [BM03]. Some of the more recent methods also support transformation from component-based modeling languages [WW04] or support multiple input languages and target performance models [WP+05]. An extensive overview of design model-based performance modeling methods can be found in [SI+04].

Detailed description of performance modeling methods is beyond the scope and purpose of this work; this overview is intended to provide an informal background for delimiting the context of the thesis and to highlight a few selected applications of performance modeling.

Architecture design

Performance-driven design of software architecture is a prominent application of performance modeling. The key idea is to assess quantitative properties of software architecture before committing to its implementation, because late changes in architecture are extremely costly and may eventually lead to failure of a project.

Designing an architecture of a software system is a complex process which is everything but straightforward. There are many decisions to make, and very few of them are black and white. Employing performance modeling during design of system architecture can increase the "contrast" of available options in a particular scenario.

Performance modeling may be difficult to use for finding the best solutions, but it can be effectively used to identify the bad solutions. Performance modeling in architecture design is not concerned about absolute performance of a system, because it depends on the final execution environment. Instead, relative performance, scalability and behavior under peak loads are the characteristics of interest. Performance modeling is therefore used to reveal potential performance bottlenecks and other performance-related problems in the system design, because performance problems originating in flawed design cannot be easily remedied by more powerful hardware and require intrusive and costly changes to be made into system architecture.

Given detailed information on the execution environment and a system with predefined performance requirements, a performance model of its architecture can be used to assess the suitability of the architecture for the system in that particular environment and provide a certain level of assurance that the architecture allows meeting the performance requirements.

The above approach forms the basis of software performance engineering [SW00] research, which is focused on construction of predictable systems. In contrast to standard development practice, performance considerations accompany a system throughout its entire life cycle, from inception and requirements specification, through design and development, to enhancement and maintenance. Performance engineering principles and techniques provide the foundation for Performance Assessment of Software Architecture (PASA), a method developed by Williams and Smith [WS02] which allows developers to determine whether particular system architecture can meet their performance objectives.

Performance prediction

Predicting system performance is a challenging, but very attractive and often demanded application of performance modeling methods. Being able to predict system performance depending on the number of users accessing system services before it is actually implemented and deployed allows making better decisions in business and operations planning.

In contrast to performance modeling in architecture design, absolute system performance is a relevant characteristic in performance prediction. However, this characteristic cannot be obtained using a performance model alone, because the overall performance of a system depends on the execution environment. The main problem is to characterize the execution environment in parameters of a particular model and modeling method.

Since modeling the entire execution environment including the system under study is infeasible, the required data need to be obtained experimentally through measurement, which in turn requires an executable artifact to be run in the execution environment.

Depending on the parameters that need to be calibrated, this artifact can be the system itself, a simplified prototype or executable model of the system, or a benchmark.

Assuming that the target system is not yet available, the question is whether to use a prototype or a benchmark. Creating a prototype can be costly and time consuming, because the prototype must capture important characteristics of the real system, while at the same time it must remain simple to justify the effort put into its development. This effort can be reduced by employing generative techniques [CGH04, DPE04] which use a system model to generate an executable prototype.

Benchmarks can be simpler to create, because they mainly exercise the environment to measure the response of the most significant operations required by the system. These operations can be identified using the system performance model and the measurements are then put back into the model. Computer systems are generally layered, both to manage complexity and to separate concerns. Typically, we can identify four basic layers: hardware, operating system, middleware, and application. A benchmark is targeted at a particular layer provides aggregate information which also reflects the influence of lower layers.

An example of such approach can be found in [LFG04, LG05] which aims at predicting performance in EJB systems. The benchmark exercises the middleware layer of the EJB, i.e. the application server. The data from the benchmark are then into a model based on queuing networks. An important aspect allowing the use of benchmarks is that the queuing networks used in the model can be solved analytically.

Similar approach can be found in [TB01] which, while focused on benchmarking of CORBA middleware, attempts to predict the outcome of the middleware benchmarks using a number of micro benchmarks exercising the operating system and elementary computational capabilities of the hardware platform. The middleware performance model combines the results of micro benchmarks using weights derived from principal component analysis of both the middleware and micro benchmark results.

1.1.2 Measurement-based analysis

Measurement-based analysis and evaluation of software systems represents the empirical branch of performance analysis research. In contrast to model-based analysis, the system under test may be identical to the system under study. For empirical analysis, there always needs to a complete computer system that performs observable and measurable actions. Given the layered construction of computer systems, the system under test may be represented by any layer; the lower layers defining the execution environment. Unless the system under test is a top-level (application) layer, there needs to be a driver application which will force the system under test to perform operations that are subject to observation and measurement.

The technical aspects of performance measurement are concerned with techniques for observing and measuring actions performed by the system under test. There is a variety of techniques applicable to different targets, under different conditions and assumptions, with different capabilities. In analogy to experimental physics, the uncertainty principle applies here as well – the more a system is observed, the more the observation influences the system and the less precise are the results.

Besides the technical aspects of performance measurement, there are also other aspects which need to be taken into account. These are mainly related to the process of measurement-based analysis and design of experiments. Experiments should have a clear

purpose, which in turn results in better understanding of requirements that need to be satisfied for the experiment to provide meaningful results.

Designing an experiment requires making a lot of decisions: how to make the system under test perform the desired operations, what observation and measurement technique to use, what kind of performance data to collect, how to derive performance metrics from the collected data and how are the metrics related to the properties of the system under study., where and how to store data, how to analyze the data and interpret the results, etc. Correct interpretation of results is especially important in performance evaluation and comparison applications of measurement-based analysis. The recommended approach is to apply the goal/question/metric paradigm [BCR94] when designing measurement experiments.

Certain measurement-based analysis techniques, such as e.g. profiling, are widely applicable and well established in practice, because they are based on concepts that can be applied in most situations. Due to their general applicability, such techniques are widely supported by different tools. However, many performance measurement techniques are often tailored to a particular application or a class of applications. Consequently, they are not so widely used in practice and because of the limited interest, they lack generic support infrastructure and tools for accessing performance data sources, collecting performance data, storing and accessing performance data for the purposes of analysis, etc.

Interestingly, such an infrastructure [SM06, HM+05] has evolved in the domain of high performance computing, where performance has always been of prime importance. The infrastructure is primarily intended to support performance analysis of high performance computing applications and their specific requirements, but in recent years it has undergone changes that make it more useful for performance analysis in general computing area.

Profiling

Profiling is probably the lowest-level measurement-based analysis technique. Without going into much detail, the principle is to observe the execution of a process and determine how much time was spent in different areas of program code. More sophisticated techniques can be used to also observe access to program data.

Observing a running program using profiling techniques results in a sequence of addresses from the virtual address space of a process. These addresses can be mapped to higher-level elements such as functions and variables of the program and libraries linked to its address space using the map of the virtual address space and program and library symbol tables. These requisites can be obtained for almost any program.

These minimal requirements make profiling a widely applicable technique, and there is an abundance of more or less sophisticated profiling tools. However, in case of large and complex systems the information provided by profiling may be too detailed to be useful. Also, traditional profiling is not very well suited for distributed systems executing on multiple nodes.

Performance evaluation

Performance evaluation is an application of measurement-based analysis to determine absolute performance of an application, subsystem, middleware, etc. in a particular execution environment. The performance of a single system can be measured in multiple

dimensions, each dimension corresponding to a particular task or an operation. Performance evaluation of multiple systems of the same type serves as a basis for performance comparison.

Performance evaluation and comparison is performed occasionally and requires a special harness to be done effectively. For the purpose of multidimensional performance evaluation, the system under test is driven by different benchmark applications to perform the operations relevant to a particular dimension of performance. The harness is responsible for executing the benchmarks and storing the measured performance data. The complexity of the harness increases when the system under test is distributed.

The results are intended for human audience and are obtained by processing and analyzing the measured data. Often this is done by humans as well. Besides the technical aspects, the most difficult part of performance evaluation and comparison is the interpretation of results. To improve the transparency of the entire process, in few cases [OMG99] there are recommendations on streamlining the process for a particular kind of system under test.

Even though different technologies and applications require different benchmarks and need to collect different data, performance evaluation is still mostly benchmark-driven and based on measurement and analysis of performance data. Consequently, performance evaluation techniques even for different systems would benefit from a shared infrastructure. Instead, numerous performance evaluation projects for different technologies have created their own supporting infrastructure [PTP00, TB01, TB06, PM+05, DOC07, CA+05, MP06], duplicating development effort despite their requirements being very similar.

Performance model calibration

Model calibration is a measurement-based technique which is used in conjunction with model-based performance analysis methods. To a large degree, performance model calibration is very similar to performance evaluation. The purpose is to obtain performance data from the system under test executing in a particular environment. The collected performance data are then processed to derive parameters of a particular model. In contrast to performance evaluation, there is no interpretation intended for human audience, but there needs to be a clear relation between the measured data and the parameters of the model.

The examples [LFG04, LG05] mentioned in the context of performance prediction use benchmarks to obtain performance metrics for operations performed by the execution environment hosting the system under study. In this particular context, the benchmarks were designed to exercise an EJB application server to determine the duration of various operations performed by the server on behalf of application components. In case of [TB01], the target of the prediction was not an application, but the middleware layer. The goal was to predict the performance of a particular CORBA middleware implementation on a specific hardware platform based on the results of low-level hardware-oriented micro benchmarks.

The issues concerning non-existing shared infrastructure which would ease the development of similar techniques for other systems are the same as in the case of performance evaluation and comparison techniques. For model calibration, the most useful feature of the supporting infrastructure would be generic access to performance data along with methods for collecting the performance data at runtime. This would ease the development of benchmarks used for model calibration.

Performance monitoring

While in case of performance evaluation and model calibration the performance data were stored away for offline analysis to provide different kind of results, the purpose of performance monitoring is to analyze and possibly visualize the measured data online, preferably in real time.

Performance monitoring is intended for overseeing complex and large scale distributed applications in real time. In case of problems occurring in a production environment, the information provided by performance monitoring can help in detecting anomalous behavior, locating a faulty component responsible for the behavior, and diagnosing the cause of the problem.

This is a step towards the concept of autonomic computing and self-healing systems [KC03]. The goal is to make the systems monitor their own behavior, and in addition to detecting and diagnosing problems to also perform corrective actions that would otherwise be performed by a human operator. The main benefit of such approach is faster recovery from known types of failures, which significantly improves the overall availability of a system [Hof05].

Also, with the shift of focus from software as a product to software as a service, techniques such as statistical process control known from manufacturing can be applied on top of performance monitoring to achieve quality attributes important for customer satisfaction, such as repeatable results and consistent reliability and responsiveness [Hof05, Rog05].

Most of the techniques used for performance evaluation can be used for performance monitoring. In addition to storing the measured performance data, the data must be delivered to a monitoring application which performs online analysis. The stored data can be used for deeper offline analysis of real world operation which can help in understanding how users access a service and provide information required for capacity planning [Hof05]. As in previous applications of measurement-based analysis techniques, a generic infrastructure for accessing and collecting performance data would ease the development of monitoring solutions.

Regression benchmarking

Regression benchmarking [BKT04, BKT05] is a technique for quantitative testing of software during development and should complement regression testing, commonly used in practice for qualitative testing. From the technical point of view, regression benchmarking is based on the techniques used for performance evaluation. However, in contrast to performance evaluation, some of the fundamental assumptions are different.

Regression benchmarking is intended to be performed frequently; at least on daily basis on a snapshot from a source code repository. The benchmarks should exercise all parts of the software in which performance is an important aspect of quality. By comparing the benchmark results of different versions of the software, changes in performance can be identified. Depending on the depth and scope of analysis, these changes can be long term, and short term. Long term changes are harder to identify, because the change cannot be identified using a small number of consecutive versions. Changes that negatively impact performance can be tracked back to the source code changes committed to the source code repository. If a problem is confirmed after analyzing the source code change, it should be fixed as soon as possible. If the source code change is correct and the change in performance inevitable, this fact should be documented.

The mainly human-driven process typically used for performance evaluation and comparison are inadequate for regression benchmarking. To make regression benchmarking a viable complement of regression testing, the entire process must be automated, including the analysis of the measured data. The amount of data collected during regression benchmarking is overwhelming, therefore the measured data are not primarily intended for human audience. Only the analysis results, and of those only results that are useful to developers.

The requirement for complete automation of regression benchmarking is a source of many technical and research challenges. The technical challenges are related to obtaining and compiling source code, executing the benchmarks and storing the measured data [KBT04]. The research challenges are related to automating the analysis of the measured data, which includes quantifying the uncertainty present in the measured data and detecting performance changes in the presence of uncertainty [KBT05a, KBT05b, KBT05c], tracing the changes in performance back to changes in source code, etc.

Regression benchmarking employs the same techniques for obtaining performance data such as those used in the context of performance evaluation. Consequently, regression benchmarking can be applied to a variety of systems and applications under development. Example applications of regression benchmarking are the MONO and TAO regression benchmarking projects [DCU07].

Like in the case of other applications of measurement-based analysis based on performance evaluation techniques, regression benchmarking would also benefit from a common infrastructure for accessing performance data sources and storing the collected data. In addition, due to its requirement for automation, regression benchmarking would benefit from a generic benchmark automation infrastructure which could be used for orchestrating all the tasks that are needed for regression benchmarking. Such an infrastructure has been proposed [KBT04] and also implemented [KL+05], but there are no experience reports yet.

1.2 Context and Motivation

Besides providing background information on performance analysis research, in the previous section we have attempted to emphasize that many methods in the area, especially those in the measurement-based branch of performance analysis, depend on availability of performance data and that there is a lack in infrastructure which would allow easily obtaining the data needed for a particular application of performance analysis. The purpose of this section is to provide an overview of the context and motivation, which led to the formulation of the problem statement and the goals of this thesis.

1.2.1 Middleware benchmarking

The Distributed Systems Research Group at Charles University has a long and extensive experience with performance evaluation, especially in the area of middleware technologies such as CORBA [TB01, BBT03] and EJB [PTP00]. In the past, the group has participated in numerous projects, both European projects and projects with industrial partners such as MLC Systeme (which later became part of Deutsche Post), Iona Technologies, Inc., and Borland International, Inc.

For each of those projects, the infrastructure for performance measurements had been written mostly from scratch; only fragments of code were reused. What was heavily

reused, however, were the key concepts related to performance measurement techniques, which were being enhanced from project to project. The concept of regression benchmarking [BKT04, BKT05] was a culmination of performance evaluation activities of the research group.

Regression benchmarking has proved, somewhat unexpectedly, to be full of technical and research challenges. However, making regression benchmarking a method, which would be commonly applied alongside regression testing, requires facing those challenges and making the process fully automated and simple to use by software developers.

Streamlining the technical aspects of regression benchmarking is also important to reduce the perceived high cost of introducing regression benchmarking into software development process. We believe the main obstacle in adopting a systematic approach to performance evaluation during software development is the apparent complexity of the whole process contrasted with difficult quantification of its benefits. Subjecting a particular application to regression benchmarking requires creating tools that exercise the application under various workloads, sample and collect performance data, process the data to obtain performance metrics, and finally analyze the results. Such tools are typically tied to a particular platform, application, or a class of applications, and cannot be easily reused in a different context.

Besides automating the regression benchmarking process [KL+05] and analysis of results [KBT05a, KBT05b, KBT05c], an important aspect which could ease the adoption of regression benchmarking as a development practice is allowing developers to use their unit tests in regression benchmarking. The tests would be used to drive the application activities while collecting performance data on important aspects of application behavior. This, however, requires a simple and generic way to access assorted performance data sources as well as infrastructure for collecting the performance data. These aspects, as emphasized in Section 1.1, are currently lacking in support.

In this context, the work in this thesis is a part of an ongoing effort at making regression benchmarking a common practice in software development.

1.2.2 Component-based applications

Besides performance evaluation, the Distributed Systems Research Group also performs active research in the area of component-based software engineering and associated formal methods for description of application behavior. Based on experience with development of a component system [PBJ98] in the early days of component-oriented research and taking into account the progress made in the area in the past five years, our group develops a second generation of the SOFA component system [BHP06].

In the past, the performance evaluation and regression benchmarking research was somewhat isolated from the component-based software engineering research. Recently, we have started to take component-based applications into account in our regression benchmarking activities, especially with respect to infrastructure support for obtaining performance data from various applications. Specifically, we have started to ask how component-based applications can make obtaining performance data easier.

In contrast to typical applications, component-based applications are accompanied by an explicit description of their architecture, including fundamental types at the level of design-elements, such as components and their interfaces. Moreover, component-based applications are constructed using the inversion of control concept, i.e. the composition of an application is controlled from outside of an application.

Our aim is to exploit these specifics of component-based applications to obtain design-level performance data for measurement-based performance analysis techniques. Having an easy access to this kind of performance data would be beneficial for most performance analysis methods which depend on collecting performance data from real systems, including regression benchmarking.

The motivation behind this approach is the idea that using the architectural information should enable creating a generic performance data collection infrastructure for component-based applications. Even more challenging is the prospect of applying this approach to heterogeneous component-based applications in which components can be implemented in different component systems or programming languages.

1.3 Problem Statement

Applications of measurement-based performance analysis techniques, such as performance evaluation, monitoring, and regression benchmarking are the key sources of information on performance aspects of a computer system. The knowledge of performance characteristics of a system under development can help in designing its architecture or in selection of components for assembly. At runtime, the information can provide insight on the behavior of an application in real-world usage and can help in resource allocation and management.

The general problem is that applying measurement-based performance analysis techniques is often perceived as too costly with hard to predict benefits for a particular project.

The perception is partially justified because of insufficient infrastructure support for different tasks, such as collecting and storing application performance data, orchestrating experiments, and analyzing the results. This situation prevents wider adoption of measurement-based performance analysis techniques, especially in software development practice.

The problem of insufficient infrastructure is associated with a wide spectrum of programming languages and platforms which would need to be supported and there is little incentive to address this issue in a systematic way.

Even in projects where there is enough expertise to apply measurement-based performance analysis techniques, the infrastructure is tailored to the specific needs of each project. This is also the case of several large open source projects such as Mozilla [MP06], Linux [CA+05], GCC [Jae05], or TAO [DOC07] which, to a certain degree, employ measurement-based performance analysis techniques to observe the quantitative aspects of their products.

The situation is slowly improving in the area of classic application development as high-profile projects such as Eclipse [EF07a] officially adopt sub-projects such as the Eclipse Test and Performance Tools Project [EF07b], which provides common tools (in the context of the Eclipse project) for testing, monitoring, tracing and profiling. Projects such as Glassbox [GBC06], Linguine Watch [Sim07], JAMon [Sou07], and possibly others only for different platforms, play an important role in advocating the practice of paying attention to performance by showing developers how to obtain performance data from their own applications. However, those are isolated projects with no common goal.

In case of more complex applications such as distributed component-based applications that are generally harder to handle, the situation is even worse. For industrial platforms,

application independent but technology and platform specific solutions appear, which solves the problem of obtaining application performance data until the industry moves on to a different platform.

There is no generic approach to collecting design-level performance data for heterogeneous, distributed component-based applications.

Even though the ease of composing applications from reusable components is often mentioned as a distinguishing feature of component-based programming, there is no easy way to take an off-the-shelf component application and obtain performance data for the application to analyze and monitor its performance, or to calibrate performance model parameters. The situation thus remains vastly identical to that of normal applications, only with increased management complexity inherent to distributed and component-based applications. Code to collect and store performance data must be written and manually integrated with a particular application.

For different applications, the work has to be repeated, even though the basic concepts related to measurement, instrumentation, experiment orchestration, performance data storage and analysis, are rather well defined and common to all platforms. Around these concepts, there is a room for an infrastructure consisting of loosely coupled modules addressing particular issues of measurement-based performance analysis, with platform specific parts encapsulated as extensions to generic functionality.

Specific requirements of a particular application of measurement-based analysis techniques would be addressed by combining the functionality of relevant infrastructure modules. Not having to create everything from scratch for every application would significantly lower both the perceived and the real costs of adopting measurement-based performance analysis techniques, especially in software development practice.

1.4 Goals of the Thesis

The general goal of this work is to reduce the perceived cost of adopting measurement-based performance analysis techniques in software development practice, specifically in the area of component-based applications which provide an explicit description of design-level elements comprising their architecture.

The goal of this thesis in particular is to propose a generic, non-intrusive and transparent approach to obtaining design-level performance data for heterogeneous component-based applications and design key parts of a supporting framework that will enable realization of the approach. Specifically, the approach supported by the framework must be:

- Generic, so that it can be applied to applications from different component systems without the need to create an application-specific performance measurement infrastructure.
- Non-intrusive, so that any modifications performed on the application to provide performance data collection capabilities do not interfere with the original implementation and do not require availability of application source code.
- Transparent, so that the entire process of performance data collection is transparent to developers, deployers, and users.

These general requirements have to be reflected in different aspects of performance data collection process concerned with obtaining and storing performance data, extending

existing applications to initiate collection of performance data related to application execution, and integrating the entire process into application life cycle.

To manage the overall complexity of addressing these aspects with respect to the above requirements, we have divided the main goal of the thesis into three subgoals, each intended to address one of the aspects of performance data collection process.

1.4.1 Subgoal 1: generic measurement infrastructure

The first subgoal is to design a generic, extensible, and portable measurement infrastructure for performance data collection. Specifically, the infrastructure is required to be:

- Generic, so that the infrastructure can be used in different situations, with different types of applications and performance instrumentation. All aspects of performance data collection should be configurable and the available performance data should be self-describing and universally accessible, without the infrastructure being explicitly aware of specific set of performance data.
- Extensible, so that relevant parts of the infrastructure can be extended to provide variety in access to different types of performance data on different platforms, or in processing and delivery of the collected data.
- Portable, so that the design concepts can be realized in any reasonable execution environment without depending on the specifics of a particular programming language or component runtime. For a given programming language, the infrastructure should provide a common base for collecting application performance data.

1.4.2 Subgoal 2: non-intrusive performance instrumentation

The second subgoal is to devise a non-intrusive, generic, and transparent instrumentation technique for use with heterogeneous component-based applications to allow collecting design-level performance data using the generic measurement infrastructure. Specifically, the instrumentation technique is required to be:

- Non-intrusive, so that extending an application to provide performance data does not require application source code and does not modify application architecture, thus making the performance data collection capability completely orthogonal to the original function of an application.
- Generic, so that the concept can be used with runtime environments of different component models. The target runtime environments may need to be extended to provide an interface necessary to support the technique, but the technique itself must not depend on features specific to a particular implementation language.
- Transparent, so that the instrumentation process is integrated with component application life cycle. The process should not require developer assistance or need to expose the application deployer to technical details associated with performance instrumentation.

1.4.3 Subgoal 3: heterogeneous application deployment

The third subgoal is to extend the deployment process defined in the Deployment and Configuration of Distributed Component-Based Applications Specification [OMG06c] to support deployment of heterogeneous component applications. Component application deployment provides an integration platform for performance instrumentation and the extension should allow using the proposed performance data collection approach even with heterogeneous component-based applications.

The choice of the OMG specification is deliberate, because it represents the most comprehensive attempt at conceptual standardization of deployment of distributed component-based applications.

Additional requirement is that the performance instrumentation process must be transparently integrated into the deployment process to avoid exposing the application deployer to technical aspects related to a particular instrumentation method.

This requirement is shared with the previous subgoal because in a distributed environment, component-based applications are executed through a deployment process which is typically far more complex than just executing an application binary on a local computer. While the properties of an instrumentation technique may provide transparency with respect to developers and the application itself, transparency with respect to an application deployer has to be achieved through integration with the deployment process.

1.5 Organization of the Thesis

The following two chapters complete the opening part of the thesis. Chapter 2 presents an overview of the most relevant works in the area. Chapter 3 discusses additional requirements on the measurement infrastructure and the instrumentation method and also provides a more detailed elaboration of the goals initially presented in this chapter.

The next three chapters represent the technical part of the thesis, each chapter providing a solution corresponding to one of the subgoals. Chapter 4 elaborates the design of the measurement infrastructure, Chapter 5 describes the use of connectors for performance instrumentation, along with a method for disabling optional connector features (including the instrumentation) to allow undisturbed application execution, and finally Chapter 6 elaborates using connectors to enable deployment of heterogeneous component-based applications in a process defined by the OMG Deployment and Configuration Specification.

The final two chapters represent the closing part of the thesis. Chapter 7 provides an evaluation with respect to the goals of the thesis and compares the presented approach with the most relevant related work. Chapter 8 then summarizes the achievements of this work and its current state, and provides an outlook at future work.

1.6 Contributions and Publications

1.6.1 Contributions of the work

The main contribution of this work is the proposed generic approach to design-level performance data collection for heterogeneous component-based applications. The technical foundation for the approach is represented by a combination of generic solutions to various aspects of performance data collection.

Since the approach targets design-level performance data related to design elements comprising application architecture, it is not suitable for performance optimization at the level of abstraction below that of the design elements. However, this can be done using many specialized profiling tools. The main benefit of the presented approach is in providing a means to obtain performance information related to interaction among components in a distributed component application, in which low-level optimizations play less significant role compared to application architecture.

The approach is thus based on utilizing the (usually) available explicit description of component application architecture. This allows collecting performance data in response to events occurring at the level of design-level elements, which in turns allows abstracting from implementation and execution specifics of various platforms, leading to a generic solution.

Due to focus on generic solutions, the proposed approach is not tied to any particular component architecture, programming language, or execution platform and provides a foundation for building technical infrastructure needed for using measurement-based performance analysis techniques with component-based applications. The approach provides an initial step towards our long term goal of reducing the perceived and the real costs of adopting such techniques in software development practice.

The solutions providing the foundation for the proposed approach are also individually useful, which allows using them separately in different context. As such they represent important secondary contributions of this thesis.

Generic measurement infrastructure

The main contribution in this area is the design of a measurement infrastructure which can be used separately to collect performance data for any application, provided that the application itself or an instrumentation layer injected into the application provides the measurement infrastructure with performance events. The infrastructure is designed to allow associating performance events with arbitrary performance data available in a particular execution environment.

The association between events and performance data can be defined by a user responsible for a particular performance measurement experiment and does not depend on particular sources of performance events. An application or its instrumentation layer is therefore only responsible for defining and emitting performance events, while a user conducting an experiment is responsible for configuring the infrastructure to collect the desired data in response to performance events. Consequently, the code responsible for emitting performance events may be very simple, which is important especially for generated instrumentation code.

One of the key objectives in the infrastructure design was to separate all infrastructure configuration operations from actual collection of performance data. This allows completely configuring the infrastructure prior to experiment and reduce possible influence of the performance data collection process on the collected data.

The infrastructure itself has a layered architecture, with configuration and management at the top, measurement operations in the middle, and performance data access at the bottom. The performance data access layer itself represents an important secondary contribution, because similar to the measurement infrastructure as a whole, the data access layer can be used separately in different context.

The performance data access layer provides the upper layers of the measurement infrastructure with unified interface for accessing diverse performance data sources. It also introduces the concept of a measurement context, which supports the objective to separate configuration and operation at this level of the measurement infrastructure. A measurement context captures a selection of different performance data that should be collected in response to operations provided by a measurement context. This is used by the upper layers of the measurement infrastructure to maintain association between performance events and performance data.

The performance data access layer itself has a low-level part which serves for accessing various platform-specific performance data sources using interfaces that may not be available in the programming language of the measurement infrastructure. The high-level part of the performance data access layer is responsible for providing the measurement infrastructure with unified view of the available performance data and for providing a unified interface for accessing the desired performance data values.

Since the mechanism for accessing performance data is designed to support different data sources, it can be extended with modules realizing access to additional data sources. This may also include modules which provide application specific performance data, or modules which provide portable portable identification for various types of performance data. The extensibility of the performance data access layer provides the upper layers of the measurement infrastructure with access to a wide range of performance data that can be associated with performance events.

Connector-based performance instrumentation

The main contribution in this area is the concept of connector-based performance instrumentation for providing design-level performance events. Using connectors for performance instrumentation in combination with the measurement infrastructure allows collecting performance data for virtually any off-the-shelf component application, provided that a connector runtime and connector generator is available for the component system used by a particular application.

Connectors are architectural elements which, along with components, comprise a component application. Since connectors are attached to component interfaces and mediate communication among components, they are a natural source of design-level performance events. Connectors allow non-intrusive integration of performance instrumentation with an application, and due to their specific place in application architecture, they provide a natural solution to issues associated with the use of proxies and wrappers. The architecture-based connector model [Bur06] used for performance instrumentation also provides additional advantages and allows encapsulating the instrumentation code into a separate entity which can be used in different connector architectures. The connector-based instrumentation process therefore only requires generating performance instrumentation code in form of a connector element which, along with other elements, comprises the implementation of a connector.

Additional contribution in this area is a connector runtime reconfiguration mechanism which allows including or excluding a particular connector element from the call path of component interface methods. This can be used to completely remove performance instrumentation code from application execution, thus letting an application execute without any (even passive) execution overhead, which is a desired feature in production environment. When desired, the performance instrumentation code can be selectively enabled at specific locations in application architecture.

Deployment of heterogeneous component-based applications

The main contribution in this area is the proposed integration of connector generation process with the deployment process defined in the Deployment and Configuration of Component-based Distributed Applications Specification [OMG06c].

Connectors are used in the deployment process to bridge technical differences among different component systems, which provides a foundation for deployment of heterogeneous component-based applications. The deployment process also provides a suitable platform for non-intrusive, connector-based instrumentation of component applications which is also transparent to the user performing application deployment.

1.6.2 Related reviewed publications

The use of a component system specific instrumentation technique together with a simple measurement infrastructure for obtaining design-level performance data without the need to modify the original application has been presented in the proceedings of the CoCoME [CCME07] project:

BB+07 Bulej, L., Bures, T., Coupaye, T., Decky, M., Jezek, P., Parizek, P., Plasil, F., Poch, T., Rivierre, N., and Sery, O. **CoCoME in Fractal**. To appear as a chapter in the Proceedings of the CoCoME Project, LNCS, Springer-Verlag, June 2007.

The work concerning the connector reconfiguration mechanism has been published in the proceedings of the 3rd European Workshop on Software Architectures:

BB06b Bulej, L., and Bures, T. **Eliminating Execution Overhead of Disabled Optional Features in Connectors**. Proceedings of the 3rd European Workshop on Software Architectures (EWSA 2006), Nantes, France, LNCS 4344, Springer-Verlag, September 2006.

The work on integrating connectors into deployment process defined by OMG in order to support deployment of heterogeneous applications has been published in the proceedings of the 1st International Conference on Interoperability of Enterprise Software and Applications:

BB05 Bulej, L., and Bures, T. **Using Connectors for Deployment of Heterogeneous Applications in the Context of OMG D&C Specification**. Proceedings of the 1st Intl. Conference on Interoperability of Enterprise Software and Applications (INTEROP-ESA 2005), Geneva, Switzerland, Springer-Verlag, February 2005.

The following publications are related to prior research on performance evaluation methods and regression benchmarking. This research has provided the necessary background and motivation which led to the formulation of the problem statement and the goals of this thesis:

KBT05c Kalibera, T., Bulej, L., and Tuma, P. **Automated Detection of Performance Regressions: The Mono Experience**. Proceedings of the 13th IEEE International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS 2005), Atlanta, GA, USA, IEEE CS, September 2005.

KBT05b Kalibera, T., Bulej, L., and Tuma, P. **Quality Assurance in Performance: Evaluating Mono Benchmark Results**. Proceedings of the 2nd International Workshop on Software Quality (SOQUA 2005), Erfurt, Germany, LNCS 3712, Springer-Verlag, September 2005.

KBT05a Kalibera, T., Bulej, L., and Tuma, P. **Benchmark Precision and Random Initial State**. Proceedings of the 2005 International Symposium on Performance Evaluation of Computer and Telecommunication Systems (SPECTS 2005), Cherry Hill, NJ, USA, SCS, July 2005.

BKT05 Bulej, L., Kalibera, T., and Tuma, P. **Repeated Results Analysis for Middleware Regression Benchmarking**. *Performance Evaluation*, 60(1-4):345-358, Elsevier B.V., May 2005.

KBT04 Kalibera, T., Bulej, L., and Tuma, P. **Generic Environment for Full Automation of Benchmarking**. Proceedings of Net.ObjectDays 2004, 1st International Workshop on Software Quality (SOQUA 2004), Erfurt, Germany, tranSIT GmbH; also in Testing of Component-Based Systems and Software Quality, LNI 58, GI E.V., September 2004.

BKT04 Bulej, L., Kalibera, T., and Tuma, P. **Regression Benchmarking with Simple Middleware Benchmarks**. Proceedings of the 23rd International Performance Computing and Communications Conference (IPCCC 2004), International Workshop on Middleware Performance, Phoenix, AZ, USA, IEEE CS, April 2004.

BBT03 Buble, A., Bulej, L., and Tuma, P. **CORBA Benchmarking: A Course With Hidden Obstacles**. Proceedings of the 2003 International Parallel & Distributed Processing Symposium (IPDPS 2003), Workshop on Performance Modeling, Evaluation and Optimization of Parallel and Distributed Systems (PMEOPDS 2003), Nice, France, IEEE CS, April 2003.

Chapter 2

Overview of the Field

The performance of a computer system can be observed at many different levels, ranging from application-level performance, through middleware and system-level performance, to cluster and grid-level performance. At each of the levels, different performance attributes may be observed, but the fundamental aspects of performance data collection remain the same regardless of the level. Consequently, the general field of performance data collection is vast and it is beyond the scope and purpose of this thesis to attempt to provide a general overview, even though there may be many similarities between works addressing performance data collection at different levels.

To provide a relevant comparison with related works, we provide a detailed description for several projects considered to be most relevant to the topic of this thesis and briefly mention other prominent approaches from different areas of computing. Later in Chapter 7 though, we only compare the approach presented in this thesis to the selected most relevant works, because we will have provided enough detail for the comparison to be meaningful.

Concerning the selection of most relevant related works, we are mainly interested in approaches to collection of design-level performance data at application-level, specifically for component-based and distributed applications.

Design-level performance data provide information related to the performance of an application as a whole, on the level of abstraction corresponding to basic and potentially dominant building blocks of such applications, i.e. components and distributed objects. Focusing on design-level performance data allows collecting reasonable amounts of performance data even for large applications, in contrast to collecting performance data related to low-level, fine grained, runtime entities responsible for implementation of a particular building block. While the low-level performance data are necessary for performing local implementation-level optimizations, they provide too much detail to be useful for analyzing application-level performance.

2.1 Component-based Application-level Approaches

Component-based application-level approaches are the most relevant to the topic of this thesis, but also the rarest. We are aware of only two approaches that utilize the properties of component-based applications in performance data collection for performance monitoring and analysis purposes.

COMPAS Framework

COMPAS [Mos04] is a framework for performance monitoring and management in enterprise applications based on Enterprise Java Beans [SUN03]. The framework consists of three main modules: monitoring, modeling, and prediction; the first two modules

representing the core of the framework, the prediction module being an extension of the core framework.

The main responsibility of the monitoring module is to obtain performance data from an application. This is achieved by adding an instrumentation layer and additional artifacts for monitoring and management to the original application to enable reporting of performance data at the level of abstraction corresponding to application components. The modeling module allows generating UML models of the application, augmented with performance data so that real values of desired performance metrics can be used instead of assumptions in application models.

Instrumentation layer

The instrumentation layer is attached to each component of the original application and consists of a proxy layer and a probe dispatcher. The proxy layer contains Probes, each corresponding to one component instance and implementing the target component's interfaces. All Probes share a single Probe Dispatcher per component class. The Probes in the proxy layer are the source of application-related events, such as method invocation and component instance creation and deletion. The events, bearing the identification of a component instance and a time stamp, are sent to a Probe Dispatcher, which is common for all Probe instances of the same type. The Probe Dispatcher processes the events, maintains a history of aggregated performance data, and potentially emits performance alerts.

The instrumentation process requires access to application metadata to determine the names of components, business interfaces, component types, life cycle methods, etc., so that a proxy layer can be generated for each component. The metadata has to be derived from application deployment descriptors, which can be obtained directly from enterprise/web application archives, or through server introspection in case of a server supporting Java Management Extensions [SUN06b] and compliant with J2EE Management Specification [SUN06a]. The required metadata has to be extracted from the deployment descriptors using a combination of EJB-specific knowledge and Java introspection.

The proxy layer is generated by expanding a *Probe Code Template* written in the Velocity [ASF07] template language. Each probe inherits from the class implementing the original component and provides its own implementation of the business interface methods which wraps the original method in the superclass with measurement code. The deployment descriptors are then altered to reference the probe classes instead of the original classes. An alternative way to create the proxy layer for a particular class is to use the JFluid [Dmi03, Dmi04] byte-code instrumentation technology, which can be used for runtime injection and removal of instrumentation code from target components.

Monitoring runtime

The monitoring and management runtime, which is also part of the modified application, is represented by a Monitoring Dispatcher. The Monitoring Dispatcher receives events from Probe Dispatchers and also provides control interface for relaying commands to the Probes. The main purpose of the Monitoring Dispatcher is to shield other clients from processing of raw events and dispatching raw commands to all Probes. Tools like Monitoring Console or Interaction Recorder are built as clients of the Monitoring Dispatcher.

The Monitoring Dispatcher interfaces with the instrumentation layer through JMX. In this context, the Monitoring Dispatcher is an event listener subscribed to JMX notifications

emitted by the probe dispatchers. On the other hand, the probe dispatchers provide MBean interface which can be used by the Monitoring Dispatcher to control the probes.

Adaptive monitoring

To reduce monitoring overhead, the Probes can be selectively switched between active and passive modes, in which a Probe either sends events immediately to its Probe Dispatcher, or buffers the events, respectively. When the buffer is full, a Probe only sends summary data to the dispatcher. The operational mode of a Probe can be either configured statically according to a monitoring profile, or it can be configured dynamically depending on the situation. To accommodate trade-offs between CPU load and communication overhead in a distributed environment, COMPAS supports two adaptive schemes aimed at reducing monitoring overhead while allowing discovery of performance problems and their origins.

The first monitoring scheme is collaborative and is carried out by the probes. The Probes are provided with vicinity information to distinguish between upstream and downstream Probes. They also have a rather high degree of autonomy and react to performance anomaly alerts from downstream probes. By communicating with neighboring Probes, each Probe can determine its own monitoring mode and detect performance problems caused by downstream Probes. The Monitoring Dispatcher is notified only when a Probe determines that it contributes to a performance anomaly.

The second monitoring scheme is centralized and is implemented by the Monitoring Dispatcher. The Probes are much less autonomous and communicate only with the Monitoring Dispatcher which analyzes the alerts received from the Probes and determines their operational mode.

To find out which probes need to be active, COMPAS uses a model-based approach [MM02] which requires knowledge about interactions among components. When the interactions are known, as long as no performance anomaly is detected, it suffices to monitor only top-level component of each interaction. When a performance anomaly alert is received, the Probes associated with the lower-level components in the interaction can be switched into active monitoring state to collect more detailed information. The model of component interactions can be obtained by Interaction Recorder, which is part of COMPAS and records events provided by Monitoring Dispatcher.

TAU CCA Tools

Common Component Architecture

Common Component Architecture [AG+99, AA+06] is a specification which aims at reducing complexity and promoting reuse of code in high-performance scientific computing applications through the use of standardized components. Because of the language diversity in the high-performance computing world, the CCA specification [CCA03] is formally expressed as a set of abstract, language independent interfaces. The interfaces are defined using a Scientific Interface Definition Language (SIDL) [DE+05], which is analogous to the CORBA IDL [OMG04], but includes support for fundamental types used in high-performance scientific computing. Besides the CCA specification, SIDL is also used for definition of CCA components. Interface and component definitions written in SIDL are then compiled using the Babel [LNL04] compiler into language specific bindings.

The CCA platform is intended to be used by developers with classic library-based application development background, which is common for the high-performance

computing environment. An application developed using CCA consists of components in form of shared libraries. The libraries are dynamically loaded into application process address at startup and their interfaces are bound together. The communication between components in different address spaces must be handled by the components themselves; CCA only provides a way to construct a single application process. This is not a problem in the domain of parallel applications, because the same process is executed on multiple nodes or processors. The communication among processes is typically based on MPI [MPI94] and the roles of individual processes and logical bindings between them are established at runtime depending on a particular computational algorithm.

In general, a CCA application consists of a launcher executable and a set of shared libraries with component implementations that are loaded by the launcher at startup. From a user's point of view, a CCA application can be started like any other application by executing the application binary. As such, it can be executed from scripts that are typically used to manage and orchestrate experiments in the environment targeted by CCA. Consequently, the CCA platform does not define any deployment process, because it is not necessary.

However, the process of launching a CCA application is still related to deployment, because it is responsible for instantiating and connecting components. The CCA runtime provides a framework-defined service called `BuilderService` which can be used to assemble and modify applications. This service is used by the launcher executable, which can be a regular program using the `BuilderService` interface, an interpreter of a scripting language which allows to direct the application assembly, or a generic launcher which can assemble an application using a declarative description.

Combining CCA and TAU

Since the execution environment targeted by the CCA platform is similar or even identical to that targeted by the TAU framework (see Section 2.3 below for an overview), it is only natural to combine the two efforts. The purpose of TAU CCA Tools [MS+05, PRL05] is to leverage the extensive functionality of TAU for measurement-based performance analysis of CCA applications.

An overview of scientific applications of the CCA platform is presented in [AA+06]. Most of the scientific applications may benefit from the performance measurement services provided by TAU CCA Tools, e.g. when used for performance profiling of component assemblies with the purpose of finding an approximately optimal set of components for execution in a specific environment [TM+06].

The TAU CCA Tools approach to performance analysis of CCA applications follows a common pattern. First, an application has to be instrumented to define measurement points and performance events; then it has to be run so that the instrumentation layer can collect performance data related to execution in a particular environment.

Component instrumentation

The application instrumentation techniques supported by TAU CCA Tools can provide performance data related to two different levels of abstraction. The first level provides white-box type performance data related to methods from behind the component interface, internal to the component implementation. This is achieved by instrumenting the component implementation code. Because of the multi-platform nature of CCA, this requires supporting various platform-specific instrumentation techniques. The TAU CCA Tools use the infrastructure provided by TAU to address this issue. The second level provides black-box type performance data related to connection between components and

thus component interface methods. TAU CCA Tools provide two instrumentation methods to provide black-box type (design-level) performance data.

The first instrumentation method of this kind uses port wrappers and is based on instrumentation of code that represents a component interface. In early C++ implementations of CCA components, such code had been written manually. Currently, the code for the desired target language can be generated from SIDL [DE+05] description of the interface using the Babel [LNL04] compiler. Since the Babel compiler does not support generating code that would be already instrumented, source code based instrumentation has to be applied on the generated code to obtain instrumented implementation of a component interface. Source code based instrumentation is supported by the TAU infrastructure; however it still requires deeper understanding of the code generated by Babel for a particular language to produce efficient instrumentation. This instrumentation method can be applied both to provided and required component ports, which allows collecting performance data both from the perspective of service provider and service user. The ability to distinguish service invocations from different users allows obtaining a whole-picture view of component interactions, which is especially important for component-based applications.

The other instrumentation method uses proxy components and is based on the man-in-the-middle principle. The proxy components provide the same ports as the original components and can be thus used to impersonate the original components where their ports are required by other components. The required ports of the proxy components are complementary to the provided ports and are used to bind the original component to the proxy component. In addition to forwarding the invocations of component interface methods to the original component, the internal implementation of the proxy component uses the performance interface to make performance measurements related to these invocations. The proxy components are generated automatically and the generator uses the Program Database Toolkit [LC+00] to infer the port and interface type information from the source code of the original component. Itself being component-based, this instrumentation method allows the proxy components to be implemented in a single language, without regard to the implementation language of the original component. However, for this approach to work, the description of application architecture and its assembly has to be modified to accommodate the placement of proxy components.

Performance measurement

The performance measurement services of TAU CCA Tools are provided in two layers. The first, lower level layer, provides basic performance measurement services that can be used by instrumented application components. These services are defined as interfaces encapsulated within a compound *performance interface* [SM+03]. The top-level interface, Measurement, is basically a factory interface providing access to implementations of other interfaces. The Event and Timer interfaces provide measurement functions for use with different performance events. The Event interface is used for recording occurrences of single events, while the Timer interface is used for recording pairs of complementary events. The Control interface is related to the Timer interface and can be used to enable and disable groups of timers. Finally the Query interface allows querying and dumping performance data associated with timers to disk. The performance interface is provided as a port of a measurement component, which can use different backend measurement systems. TAU CCA Tools provide an implementation of the performance component on top of the TAU framework.

The second layer uses the performance component to provide higher-level services needed for performance monitoring. These services are implemented by a *Mastermind* component [RT+04], which handles runtime collection, storage, and reporting of performance data. The *Mastermind* component is used by the proxy components which report performance events by means of a MonitorPort interface. The events are related to method invocations performed on component interfaces intercepted by a particular proxy. The *Mastermind* component stores performance data for each monitored method and provides management functions that allow enabling and disabling the collection of performance data for a particular method.

2.2 Middleware-based Application-level Approaches

Middleware-based approaches are similar to component-based approaches in they can be also used to obtain design-level performance data related to execution of a distributed application. In contrast to component applications though, design-level entities are represented by distributed objects.

Wabash

Wabash [SMM00] is a tool for non-intrusive testing, monitoring and control of distributed applications. The approach employed in Wabash can be applied to broker or RPC-based applications built on OMG CORBA, Microsoft COM, SUN RMI, or RPC middleware in general, however Wabash itself only supports CORBA-based distributed objects.

The architecture of Wabash is intended for large-scale distributed applications built of objects distributed over a wide geographical area. In such applications, objects are expected to form clusters of collocated objects, with interconnections between the clusters being realized by a limited set of boundary objects. The clusters of objects correspond to logical *zones* which define the partitioning of the geographical area. An application is therefore viewed as a hierarchy with first-level nodes representing zones, each zone containing machines with server objects. Objects deployed in each zone are managed by a *zonal manager*.

The features of Wabash include performance measurement of individual server objects, displaying application architecture at runtime, method and interface coverage information, fault injection, and event-based monitoring and control. To use Wabash with a distributed application, the application needs to be first instrumented to provide a *LocalListener* layer for each server object. Application instrumentation is realized using CORBA interceptors, with each interceptor representing an instance of a *LocalListener* corresponding to a particular server object. All requests to a particular server must go through its *LocalListener* instance. At runtime, each *LocalListener* instance registers with its corresponding zonal manager and the Wabash GUI uses them to manage the application and provide the above features.

Each zonal manager consists of several parts, each responsible for a particular feature of Wabash. The performance measurement and control part is based on earlier work [SMD00] and besides support in the zonal manager, each *LocalListener* contains a *Performance Instrumentation module* which and allows collecting memory usage and basic statistics such as minimum, average, and maximum latency associated with individual methods of server interfaces.

The functionality of each Performance Instrumentation module instance is realized in three different contexts. The main context corresponds to the context in which arriving

requests are serviced. An incoming request is intercepted and the interception code time stamps the request and takes a snapshot of various process attributes before letting the request reach the server object. The response from the server is also intercepted and the interception code completes calculations of request processing time, actual processor time used, and memory consumption attributes related to a particular request. After updating statistical information, the interception code posts an event to a queue which activates threads in other contexts.

The events are processed asynchronously by a Performance Agent thread along with another thread logging the events for offline analysis. The Performance Agent provides aggregate statistics to a *Monitoring and Control Module* of the zonal manager and also allows the manager to selectively enable or disable performance measurement for a particular server object. Collecting performance data other than the mentioned statistics and memory usage is not supported.

StatWrapper

Similar to Wabash and its server-object performance measurement, the approach proposed in [MC+99] is aimed at collecting metrics for CORBA-based distributed systems. This is done by adding a performance measurement layer to existing server-object implementations. The measurement layer, referred to as *statWrapper*, provides metrics related to number of messages received by an object, the amount of data transferred, total service time, server utilization, etc.

The proposed approach is primarily focused on data collection and in this respect it mainly elaborates the instrumentation techniques suitable for general application to CORBA objects with emphasis on non-intrusiveness to avoid modification of existing server-object implementations the source code of which is assumed to be unavailable. This is related to both extending the functionality of the original server-object implementation as well as extending the object interface so that it remains accessible from older clients but also provides access to the extended performance measurement functionality.

Most discussion is dedicated to analyzing the compatibility of inheritance-based and delegation-based object wrapping techniques with CORBA-specific application of these techniques to implementation of server objects (BOA implementation inheritance, TIE class delegation) and techniques for adding new methods to server-object interfaces. With the exception of enumerating the performance metrics relevant to CORBA-based systems, details concerning the internals of the *statWrapper* layer related to access and collection of performance data are not discussed.

Other middleware-based approaches

OrWell [WK98] is a monitoring environment for distributed object-oriented applications written using the ObjectWire communication framework. OrWell allows dynamically attaching special entities called *Event Distribution Plugins* to applications written using the ObjectWire framework. The EDPs then provide OrWell analysis tools with events related to communication among different parts of a distributed application. The analysis tools then use the events for dynamic analysis of various aspects distributed applications, including aspects related to application performance. Even though OrWell uses an outdated middleware platform, the basic concepts can be implemented e.g. using interceptors in context of CORBA middleware, with the exception that interceptors cannot be registered dynamically with an ORB instance.

MODIMOS [ZL+95]is a system for monitoring object-based distributed applications. The main purpose of the system is to provide structural and behavioral visualization for heterogeneous applications built of parts written using different distributed computing environments. The architecture of MODIMOS consists of several layers. Besides presentation layer, there is a global monitoring layer which, through an interoperability layer, manages a set of local monitors corresponding to monitored applications in different computing environments. Monitored applications are instrumented to provide the local monitor with events related to different abstraction units, ranging from methods and interfaces to a distributed computing environment. MODIMOS supports various distributed computing environments including CORBA, but we have been unable to find any recent information about the project.

2.3 General Application-level Approaches

General application-level approaches are typically intended for performance analysis and optimization of "normal" applications in which design-level entities typically correspond to implementation objects or libraries. Even though the performance analysis process remains the same and comprises application instrumentation, performance data collection, and performance data analysis, these approaches are more suitable for performance optimization because they provide data related to low-level implementation entities.

In contrast to component or middleware-based applications, "normal" applications in general do not provide common design-level entities that could be used to provide a high-level overview of application performance.

Consequently, we do not consider general application-level approaches to performance data collection to be among the works most relevant to the topic of this thesis, with the notable exception of the TAU performance system, which is used to provide performance data collection capabilities for the component-based approach of TAU CCA Tools.

TAU Performance System

Tuning and Analysis Utilities [SM06, PRL07a] is a tool framework for performance evaluation, analysis, and tuning in high-performance computing applications. The framework is multi platform, supports programming languages commonly found in the high-performance computing area but also languages such as Java, which are more common in application programming domain. The TAU architecture consists of three layers, with specialized tools and modules in each of the layers. Through adherence to well-defined interfaces and data formats, the TAU framework is very flexible and can be configured to suit the needs of a particular user.

The first layer is responsible for performance instrumentation. The TAU instrumentation model and associated tools allow inserting instrumentation code into applications almost in any form, ranging from source code instrumentation, through object code patching, wrapper libraries, binary code patching, to virtual machine-based instrumentation [SMA01]. The instrumentation code is responsible for definition of performance events. In this respect, TAU provides several predefined event types, such as system, library interface, and code location events, as well as user-defined events. Each event is assigned an identifier which is included in calls to the *TAU measurement API* when reporting performance events.

The second layer is responsible for performance measurement and consists of five basic blocks: event management, profiling, tracing, performance data sources, and OS and runtime system. The *event management* block is used by the instrumentation code to provide the measurement layer with performance events. The *performance data sources* block provides an interface to various time sources, hardware counters, and system performance data. The *OS and runtime system* block provides allows TAU to adapt to the underlying execution platform. The most important blocks are the *profiling* and *tracing* blocks, which provide support infrastructure for two different measurement modes. Profiling is concerned mainly about collecting aggregate performance metrics related to individual performance events and TAU supports several profiling modes in this respect. Tracing is concerned about collecting traces of performance events during program execution, mainly for offline analysis. The support infrastructure in both blocks supports sampling, processing, storage, and output of performance data.

The third and last layer is responsible for data analysis and visualization. For data resulting from each measurement mode, it contains tools for translating between different profile and trace formats used by various visualization tools. Profile data can be stored in a common data format using the Performance Data Management Framework [HM+05], which provides unified access to profile data from various systems.

High-performance and parallel computing approaches

The area of high-performance and parallel computing contains (quite understandably) many works related to collection of performance data for measurement-based performance analysis. A report [RCJ00] from the first phase of the APART project provides an overview and basic categorization of 10 different tools (as of February 2000) that assist in automating performance analysis of parallel programs typically written in C, C++, or Fortran. Today, the number of such tools will be even higher, but we have not tried to compile an updated list. Instead, we have taken the liberty of mentioning (in our view) the most prominent projects in this area.

Paradyn [MC+95] is a measurement tool for parallel and distributed applications which can be used to search for performance bottlenecks in long running applications executing in a large-scale distributed environment. Paradyn uses dynamic patching of machine code to place performance instrumentation into an application and allows collecting operating system, hardware, and application specific performance data. A notable feature is allowing users to limit the instrumentation overhead imposed by performance data collection on application execution.

KOJAK [MF03] is a tool set for automatic performance analysis of parallel applications, which includes instrumentation, performance data collection, and analysis and visualization of the collected data. In addition to method-level instrumentation at source code or binary code level, KOJAK also supports source-level instrumentation for OpenMP constructs. The instrumentation is primarily intended to provide method-level event traces for postmortem analysis. Additional performance data obtained from processor hardware performance counters can be associated with the events. Event traces resulting from execution of instrumented applications can be processed by a specialized tool for automatic trace analysis and presentation of results.

SCALEA [TF03] is one of the parts of the *ASKALON* [FJ+05] tool set and is responsible for instrumentation, measurement, and performance analysis of parallel programs written in Fortran. It consists of an instrumentation system, runtime, and performance analysis and visualization system. The instrumentation system supports source-based instrumentation

including OpenMP constructs. The runtime system used by instrumentation code allows collecting performance data related to regions of code in the instrumented program. Performance data associated with instrumentation events may consist of timing information and contents of hardware performance counters. The analysis part calculates additional performance metrics from event traces and allows visualizing the results.

2.4 System-level approaches

In contrast to application-level approaches, system-level approaches are targeted at performance data collection related to entire computing systems, ranging from single computers, through LAN-based clusters, to computational grids.

LeWYS [OWC04] is a component-based system monitoring framework written using the Fractal component model. It uses a system of probes for accessing various sources for system-related performance data, and a component based framework for transporting the data to consumers. Using the *DREAM* [OWC07c] communication middleware, LeWYS creates a system of monitoring pumps which periodically initiate sampling and transfer of system-wide performance data.

Cluster and grid computing approaches

The situation in the area of cluster and grid computing is similar to that of the high-performance and parallel computing domain, except there are even more tools available. An overview of grid performance monitoring and evaluation tools [GW+04] lists and categorizes 26 different tools in 6 dimensions corresponding to target audience, functionality, features, instrumentation, tool architecture, and available interfaces.

Since the relation of those tools to the topic of this thesis is even weaker than that of general application-level approaches, we have not attempted to study the tools in more detail.

Chapter 3

Requirements and Goals

The goals presented in Section 1.4 only capture general requirements on various aspects of the solution. To prepare ground for elaborating the goals in more detail, we provide an overview of key requirements for performance data collection along with specific requirements arising from the goals of this thesis. Using this as a foundation, we reiterate through the goals of the thesis, putting them in a more specific context and providing specific objectives for each of the subgoals.

3.1 Performance Data Collection Requirements

Application of measurement-based performance analysis techniques generally requires a rich infrastructure, which provides support for experiment management and orchestration, performance data collection, persistent data storage, and data analysis. Even though all aspects of such support infrastructure are important, the most important is the ability to obtain performance data for a system under test, which subsequently enables applying various measurement-based performance analysis methods on the system.

The approach to performance data collection is also the most volatile part of such infrastructure, because it is highly dependent on the character of the system under test. Given the context and scope of this thesis, we will focus primarily on this aspect of measurement-based performance analysis and the specifics of obtaining performance data for heterogeneous component-based applications.

3.1.1 Knowing what data to collect

We have already used the terms *performance data* and *performance metrics* in previous text, but so far the distinction between the two was not really important. For the text that follows, though, the distinction should be clear.

Performance data is a collection of values obtained by recording observations of various attributes during execution of a system under test. The attributes need not be directly related to the system under test; they may be related e.g. to the execution environment, hardware resources, etc. Performance metric, on the other hand, is a performance characteristic with a particular semantics directly related to the system under test. In many cases, performance metrics can be observed directly, but in other cases, performance metrics must be derived from performance data. For simple performance metrics, this can be done at runtime, for more complex metrics, offline analysis of performance data is required.

As mentioned earlier in Section 1.1.2, an important prerequisite for applying measurement-based performance analysis techniques is having an experiment with clear purpose. This means that we have to specify what we want to know about a system under

study and how obtain this information from experiments performed on the system under test. This typically requires identifying performance metrics that are necessary to synthesize the desired information. And since many performance metrics cannot be obtained directly in form of performance data, it is imperative to know what performance data need to be collected to obtain the required performance metrics.

Performance data are always associated with events occurring in a system under test, because these events trigger the collection of performance data, which is done by reading and recording values of desired attributes from various sources. Conceptually, the events are defined by a set of conditions which, when satisfied, signify the occurrence of an event. In a real system though, the conditions for system events have to be explicitly evaluated and the collection of performance data has to be explicitly initiated in response to relevant events.

Collecting performance data for a system under test thus requires the ability access and record values from various data sources, which is provided by a measurement infrastructure, and the ability to initiate the collection of performance data in response to various events, which is provided by performance instrumentation.

3.1.2 Accessing performance data sources

Application performance metrics are typically derived from raw performance data, which can be obtained from various sources in form of performance counters and gauges. Consecutive readings of a single counter are required to be monotonous and non-decreasing, with the exception of counter overflow, which must be taken into account. Gauges, on the other hand, provide absolute readings for a given moment and there are no restrictions on the nature of changes between multiple readings other than those implied by the semantics of a particular gauge. A typical use of counters is to track number of events or quantities, such as number of bytes transmitted over a network interface, while gauges are typically used to convey information about consumable resources, such as the amount of memory available, or system utilization.

The availability of raw performance data sources is highly dependent on the execution environment. The sources of performance data are very diverse and can be found practically at any technology level of a computer system.

The lowest level performance data sources can be found in the hardware, i.e. the processor and various hardware devices. Modern processors are equipped with performance counters that allow counting events such as cache misses, instruction fetches, retired instructions, clock cycles, etc. Hardware devices such as network interface cards provide counters for transmitted and received packets, collisions, etc.

The middle level performance data are provided by an operating system. The performance data an operating system provides are associated with the resources it manages, e.g. processor time, memory, network bandwidth, storage capacity, etc. An operating system also often provides a more-or-less unified access to performance data from hardware devices.

The top level performance data sources can be found in applications. Even though from the view of an operating system an application is just a process executing in its virtual memory, the internal architecture of an application is often a complex structure of layers, subsystems, and other logical blocks supporting the application logic. Each of the logical blocks is a potential source of performance data related to the function performed by a particular block.

Performance data from lower technology levels are less application and domain specific and more difficult to obtain in a generic way from applications written in high-level programming languages or executing in a virtual machine. This is caused by increased diversity of methods required for accessing the data, because different middleware layers and operating systems have different programming interfaces for accessing performance data, different processors have different performance counters and require different instructions to access them. Accessing low-level performance data often requires using APIs that are only available in low-level languages.

The main requirement on the measurement infrastructure is to bridge the diverse sources of performance data and provide unified access to performance data from different technology levels.

Performance data sources

Processor performance counters

PAPI [BD+00, PAPI] is a library for accessing hardware performance counters found in modern processors. Besides a convenience layer which streamlines the sampling and management of performance counters, the library also provides a system of generic performance counter names, which can be used to access the most common counters such events such as retired instructions, cache misses, etc. in a processor-independent way. The library has been ported to different operating systems and contains also bindings for the Java language.

PCL [BZ98, BZM03] is a library with the same purpose as PAPI, but with simpler interface and generic event names. Even though it is still used in various projects, it is not actively maintained anymore and has been superseded by the PAPI library.

Performance data provided by operating system

With the exception of processor performance counters, most useful performance data is typically provided by an operating system. This includes performance data related to both physical parts of a computer system, such as network interface cards, or hard disks, and logical entities, such as network stack, TCP connections, time spent executing individual user processes, etc. There is virtually no difference in the type of data provided by various operating systems. However the method for accessing the data is different for each operating system.

The Linux operating system does not provide a comprehensive, system-wide interface for accessing device and system specific performance data. Most information that can be related to system performance data has to be obtained by specialized system calls, communication with kernel subsystems, or by reading files from the *proc* and *sysfs* virtual file systems. While the last method is typically used the most, it has drawbacks in that it provides data in ASCII form which needs to be converted to binary representation after each reading. Moreover, the performance data are scattered across multiple files therefore obtaining the desired data may require multiple system calls.

The Windows operating system provides a comprehensive, system-wide interface to access all performance data managed by the operating system. Moreover, the interface allows accessing performance data from remote machines and also allows applications to register with the system to provide their own performance data. The performance data can be accessed through *Performance Objects*, with each *Performance Object* containing a set of named performance counters, with multiple instances in case of *Performance*

Objects corresponding to logical or physical entities occurring in the system multiple times, such as performance objects corresponding to user processes. The Windows Performance Data interface is binary and is accessed through the system registry. Compared to Linux, it allows obtaining the necessary data in less system calls, but does not allow requesting individual Performance Object instances or specific counters, which in case of certain performance objects requires significant amounts of memory to obtain even a single counter. The performance data are encapsulated in self-describing binary structure that has to be navigated after each sample.

The Solaris operating system provides a combination of the approaches found in Linux and Windows operating systems. It allows accessing performance data using specialized system calls or through a *proc* file system, but it also provides a binary interface to performance data managed by the operating system kernel. The kernel interface can be accessed through a `kstat` library which communicates with the kernel using the `/dev/kstat` device. The data are provided in a list with each item defining its type and reference to type-specific data, therefore the data are not self-describing as in case of the Windows operating system. On the other hand, the library interface provides access to data associated with individual instances of performance entities identified by class, module, and instance name or identifier. In addition, the structure only needs to be parsed once; the data are updated in place.

Performance data provided by management frameworks

Above operating systems, there are technologies such as SNMP [RFC1157] and JMX [SUN06b] which are intended for observation and management of various entities in a system. The much older SNMP standard has been originally designed for network devices, to allow managing various network devices using a generic protocol, which can be used by monitoring and management consoles. SNMP may be also used by applications to expose performance data and management interface.

The JMX technology is primarily associated with the Java language, and is intended to allow applications to expose a monitoring and management interface for use from generic management applications. Since the JMX technology is primarily intended for software, it uses a purely user defined naming system for management objects and is generally more flexible than SNMP.

Requirements

There are desirable features in several of the presented technologies, yet none of the technologies contains the right combination of features to satisfy our requirements.

The requirements related to collection of performance data can be split into three groups. The first group of requirements is related to the ability of the measurement infrastructure to access various data sources, the second group is related to the runtime management of the collected data within the infrastructure, and the third group contains general requirements.

Access to different data sources

There are many sources of performance data related to different parts of a computer system, but for each data source a different interface has to be used to access the data it provides. Concerning access to different data sources, the measurement infrastructure must satisfy the following requirements:

- The infrastructure must provide an abstraction for accessing high-precision time sources available on different platforms.

  Technically, a time source is basically a performance data source, but it is the most important because without timing information, any other performance data lose their meaning.

- The infrastructure must be extensible to allow providing performance data from different parts of a computer system such as the processor, operating system, middleware, virtual machine, etc.

  This is best seen in case of Windows Performance Data and the management technologies based on SNMP and JMX. All of them can be extended to provide access to additional data sources.

- The infrastructure must provide a unified interface for accessing performance data.

  Again, this is best seen in case of Windows Performance Data and the management technologies based on SNMP and JMX. Each of the technologies defines a performance data model which must be supported by all registered data sources.

- The infrastructure must support efficient data acquisition to minimize the overhead associated with collection of performance data.

  This, in contrast, is worst seen in the case of Windows Performance Data as well as the management technologies based on SNMP and JMX. Windows PD subsystem provides only a coarse-grained access to performance data, SNMP and JMX were primarily designed for collection of performance data in a distributed environment, over a network.

- The infrastructure should allow accessing semantically equivalent performance data in a generic and platform independent way.

  This is best seen in case of the PAPI library which supports generic names of semantically equivalent performance counters.

Runtime configuration, management, and control

Besides the ability to access different sources of performance data, the measurement infrastructure is also responsible for sampling, processing, and storing the performance data in memory, delivering the stored data to consumers, and coordinating the configuration and measurement operations of the infrastructure as a whole.

The above mentioned data sources are mainly supposed to provide access to specific kind of performance data, and with the exception of the PAPI library, they do not provide any runtime data management, which requires high-level functions that are often better left to the users of the technologies. The PAPI library provides a very limited support for managing the collected data, but also leaves the high-level functions at the discretion of a user.

Including high-level functions in such technologies would be counterproductive, because it would limit their flexibility. However, the purpose of the measurement infrastructure is to enable unified access to various performance data sources and to record the data collected in response to performance events. The measurement infrastructure must therefore satisfy the following requirements:

- The infrastructure must support runtime configuration, which should allow selection of events that trigger collection of performance data and association of different types and instances of performance events with collection of different types of performance data.
- The infrastructure must support efficient storage of the collected performance data in memory along with configurable policies determining the behavior of the infrastructure when the allocated storage has been exhausted.

  For many applications, passing the performance data immediately to a consumer would be highly inefficient and result in a considerable overhead. Therefore the infrastructure has to buffer the data until they are delivered to the consumer, or discarded according to a configured policy.
- The infrastructure should support delivering the performance data to consumers according to a configured policy.

  This is basically an optional extension of the requirement for storing performance data. The purpose is to avoid exhausting the data storage capacity by delivering the data to the consumer at runtime.
- The infrastructure should support limited immediate processing of performance events and associated data to derive simple performance metrics and aggregate information.

  This allows significant reduction of the amount of data that needs to be stored and transported in applications where only simple or aggregate performance metrics are required.

General requirements

The measurement infrastructure being a kind of middleware, the intention is to have a native implementation of the infrastructure for each programming language, possibly with special modules written in other languages, providing access to performance data sources using an interface not available in the target programming language of the infrastructure. However, even then the infrastructure should remain portable:

- The implementation of the infrastructure in a particular programming language must be portable to support different operating systems and hardware platforms.

  This is worst seen in case of the operating system specific methods for accessing performance data. While generally well designed and hardware agnostic, their major drawback is that they cannot be used in portable applications.

3.1.3 Instrumenting applications for performance measurement

The main responsibility of performance instrumentation is to define performance events [SM06] and deliver those events to the measurement infrastructure to initiate collection, processing, and storage of performance data. Event semantics defines which conditions in application execution lead to occurrence of an event. The placement of instrumentation code in an application reflects the semantics of events it is able to generate. The set of event types supported by an instrumentation layer restricts the level of information that can be determined about a system by recording event traces.

The most primitive event generated by the instrumentation layer has to provide the semantics, location, context, and a time stamp; in other words, what happened, where, to who, and when. An example is an event which informs an event listener that a particular thread entered or left a particular business method of a particular component.

In most cases, the desired information or performance metrics concerning a system can be determined at runtime from the semantics and the data associated with a single event. The level of information that can be obtained from single events depends on the semantic level of events provided by the instrumentation layer. Information beyond the semantic level of single events needs to be determined using information from multiple events, which may be difficult to perform at runtime. In such cases, the desired information has to be determined by offline analysis of recorded event traces.

This can be addressed by using groups of semantically related events, such as e.g. a pair of complementary events emitted when entering or leaving a particular method, or events with more complex semantics. Such events provide more high-level information that can be inspected at runtime.

The additional information associated with more complex event semantics can be also used to reduce the space needed to store event records. Complementary events can be merged into a single event with a slightly different semantics, containing the information originally associated with the two separate events. Such an event does not allow reacting separately to the original events, but it can be recorded using only slightly more than half of the space needed for the two events. For example, merging the events for entering and leaving a method results in a single notification event emitted after a method call, containing both the method invocation and return times originally associated with separate events.

There are many ways to integrate performance instrumentation code with an application in order to define and emit performance events. However, in context of heterogeneous applications, the main requirement on application instrumentation is to provide a method that is applicable to components from different component systems intended to run on different platforms.

Instrumentation techniques

Source code modification

Modifying application code to include performance instrumentation is one the basic instrumentation techniques. An obvious disadvantage is the dependency on the availability of application source code; therefore this technique is typically used internally by application developers. Source code instrumentation allows the instrumentation code to become an integral part of the application, which results in slightly lower instrumentation overhead than with other techniques.

Source code instrumentation can be performed by simple tools by e.g. replacing simple annotations in application code, by using source code manipulation tools, such as the Program Database Toolkit [LC+00], or by creating instrumentation-friendly applications using libraries such as DPCL [DHH01] which allow a developer to wrap e.g. application objects in code enabling dynamic instrumentation, etc. In addition, aspect oriented programming methods and tools, such as AspectJ [EF06], can be also used to instrument an application, enabling to use the expressive power of *pointcuts* and *joinpoints* to define executing context in which the instrumentation code should be activated, which can be used to define performance events with complex semantics.

Binary modification

Instrumentation based on modification of application binary is probably the most used technique when the application source code is not available. The modification can be performed either directly on application binary, using symbol information contained in the binary, or at runtime, patching the application code after it has been loaded into memory. Binary modification cannot be done manually and tools and libraries such as the DyninstAPI [BH00] have to be used to perform the insertion of instrumentation code into an application.

Binary modification can be also performed on platforms based on interpretation of intermediate, platform independent code, such as the Java byte code. Compared to modification of native binary code, byte code manipulation is hardware independent, and therefore represents a more acceptable approach to instrumentation. In the case of Java byte code, libraries such as ASM [OWC06] or BCEL [ASF06] can be used.

Structural modification

Another class of instrumentation techniques is based on modifying the runtime structure of an application, and is typically used in object oriented or component based systems. This technique can be used only for instrumentation at the granularity of method invocations, because the basic principle is to use proxies and wrappers to impersonate other entities in method-level interactions at runtime. Both proxies and wrappers provide a facade compatible with the type of the impersonated entity and delegate method invocations to the original entity.

Proxies and wrappers are typically created manually or generated by a specialized tool. On platforms with support for runtime introspection and reflection, proxies can be even created at runtime, which is used e.g. by the SUN implementation of RMI.

In contrast to previous methods, this type of instrumentation cannot be used generally, and depends on application specific approach to replace the original entities with proxies and wrappers.

Indirect instrumentation

Indirect instrumentation techniques are suitable for interpreted or virtual machine based environments. The instrumentation is typically based on attaching user defined code to the interpreter or virtual machine and registering callback methods for significant events such as method invocations or variable assignments.

The types of available interpretation events as well as the interface for attaching user-defined code to an interpreter or a virtual machine are highly platform dependent. An example of a virtual machine interfaces suitable for performance instrumentation is the Java Virtual Machine Profiling Interface (JVMPI) or the more recent Tool Interface (JVMTI).

On the other hand, indirect instrumentation is non-intrusive and completely transparent to an application. Moreover, the instrumentation can be attached and detached at runtime, allowing an application to run without instrumentation unless required.

Indirect application instrumentation can be also achieved using the *DTrace* [CSL04] framework of the Solaris operating system. The framework provides a unified interface to a number of instrumentation providers, which define instrumentation points primarily in the operating system kernel, but also in system-level applications such as a Java virtual machine. User-defined activity may be associated with each instrumentation point, with the activity described in a C-like language without loops. Along with data aggregation

capabilities of the DTrace framework, this allows creating application-specific system-wide instrumentation, which may be also used for collecting performance data related to externally visible application activities.

Requirements

There are many techniques for instrumenting applications in general. However, not all of them are suitable for instrumenting heterogeneous component-based applications. The technique envisioned for the approach presented in this work has to satisfy the following requirements:

- The instrumentation technique must allow providing performance events to drive performance data collection at the level of abstraction corresponding to design-level application elements, such as components and their interfaces.
- The instrumentation technique must be generic so that it can be applied to heterogeneous component-based applications without the need to apply different techniques to different components depending on their target execution environment and programming language.

  This implies that the fundamental concept behind the technique must not be specific to a particular platform or a programming language, which would prevent implementation of the concept in component systems based on different platforms and languages.
- The instrumentation technique must be fully automatic so that it does not require developer assistance.

  This is a fundamental requirement, because besides being able to inject instrumentation code into specific places, there must be instrumentation code to inject. All instrumentation code and associated artifacts must be therefore generated by a machine.
- The instrumentation technique must be applicable with only application binaries and metadata describing the application architecture and assembly being available.

  This is important to allow applying the proposed performance data collection approach even (or especially) to off-the-shelf component-based applications or applications using third party components for which the source code is unavailable.
- The instrumentation technique must be integrated with the deployment process to allow transparent instrumentation of component-based applications.

  This requirement is related to the automation requirement, though in this context the purpose is to avoid exposing application deployer to technical aspects of the instrumentation process, such as requiring the deployer to modify the application assembly description to enable the instrumentation.
- The instrumentation technique should provide control over the execution overhead imposed on the instrumented application.

  This is especially important in production environment, where the ability to manage instrumentation overhead allows retaining the instrumentation

code even in production-level applications for on-site diagnosis and monitoring.

3.2 Goals of the Thesis Revisited

As a summary of the requirements, the following sections provide a more specific description of the goals presented in Section 1.4. The original structure of subgoals has been retained, but the individual subgoals have been elaborated to reflect the requirements from Section 3.1 and various aspects of selected related works discussed in Chapter 2.

3.2.1 Measurement infrastructure

The measurement infrastructure has three main responsibilities: providing access to performance data, sampling and storing the performance data, and delivering the data to a consumer. On top of that, the infrastructure should be configurable at runtime, because it is expected to be used for monitoring applications and diagnosing problems in production environment. Therefore it must allow a user to define what to measure, where to measure it, and what to do with the data as well as provide runtime control over a infrastructure operation.

Unified access to performance data

The sources of performance data are very diverse and in execution environment, there may be different performance data sources. The availability of specific performance data sources depends on hardware architecture, operating system, middleware technologies, application frameworks, and may potentially include the application itself. The infrastructure should be able to provide a unified view of the performance data available in various data sources and access the data in a unified manner, regardless of their origin.

This aspect of the infrastructure should be addressed by a specific subsystem providing a unified interface over different, typically low-level, interfaces for accessing performance data in various sources. Since the low-level interfaces may not be available in the implementation language of the measurement infrastructure, language-specific bindings should be used to access the low-level interfaces from within the subsystem.

To reduce overhead associated with obtaining samples of performance data, the infrastructure should allow collecting a predefined set of performance values in such a way so that the lower layers responsible for accessing the performance data sources can select an optimal data retrieval strategy.

The objective is to design a performance data access layer providing the measurement infrastructure with a unified interface for efficient access to performance data from diverse sources.

Collection and storage of performance data

The measurement infrastructure is expected to obtain the values of performance data from selected data sources in response to performance events provided by application instrumentation. Even though in this work we are mainly interested in design-level performance events corresponding to invocation of business methods on component interfaces, the collection of performance data can be generally associated with any kind of

events, such as e.g. expiration of a timer. The infrastructure should therefore support associating and collecting performance data with generic performance events.

The measurement infrastructure is also responsible for storing the data associated with performance events in memory until they are delivered to a consumer. However, even when no performance data are associated with an event, a performance event itself may still provide information, such as the time of event occurrence, which is the most basic form of performance data needed for performance analysis. Therefore the infrastructure has to store the event data along with optional performance data from various data sources.

In addition to managing storage of event records, the infrastructure may support online transformation and aggregation of performance data in order to reduce the amount of data that need to be stored or transported over a network. However, any processor intensive computations should be left for offline analysis to avoid influencing the execution of an application by excessive load on the processor.

The objective is to design a measurement infrastructure layer for processing performance events, collecting performance data associated with performance events, and storing event records along with associated performance data using memory-based storage to minimize the influence of performance data collection on application execution.

Configuration and control over data collection

When using the performance data access layer providing generic access to performance data, the measurement infrastructure should support dynamic association of performance data with performance events. The infrastructure should therefore provide means to enumerate available events as well as available performance data and allow an external entity to initiate the association.

To manage the overhead imposed on the executing application by performance data collection, the infrastructure should provide a means to configure which performance events should be emitted by application performance instrumentation. An alternative approach based on ignoring performance events within the infrastructure would provide similar functionality, however to minimize the passive overhead of unused performance events the infrastructure should communicate with performance instrumentation entities to disable emitting particular events within the instrumentation.

The objective is to design a configurable measurement infrastructure which allows external entities to dynamically associate performance events with performance data and provides them with mechanisms to enable and disable individual performance events.

Performance data delivery

A consumer of the performance data collected by the infrastructure may be a monitoring application, a performance data repository, or any other application. The methods of delivery can be very diverse, ranging from storing the data into files on local nodes, to sending it over the network using push or pull based methods.

Related projects such as COMPAS [Mos04] or LeWYS [OWC04] implement methods for transferring performance data to the consumers. COMPAS uses JMX to publish events, which are delivered to consumers subscribed to receiving the events, while LeWYS uses the DREAM [OWC07c] framework to create a system of data pumps to transport data. However, in both cases the transport method is fixed in the design, which limits the approach to a particular environment and usage scenario. This is not necessarily an issue,

because both target a specific platform (enterprise Java applications in case of COMPAS) or have a specific purpose (system monitoring in case of LeWYS).

The goal of this work is to provide a generic approach to performance data collection for component-based applications and the challenge lies in making the approach applicable with different component systems and in different runtime environments. In contrast, the particular method of delivering performance data to consumers depends on the intended use of the data and not on the properties of the component or other applications the infrastructure is actually used with. Properly addressing this aspect of the monitoring infrastructure would require analyzing the typical use case scenarios in the context of activities briefly introduced in Section 1.1 and design an efficient data transfer method for each of them, which we consider beyond the scope of this thesis.

Consequently, we do not explicitly address this aspect of the measurement infrastructure in this work. Instead, the objective is to design the measurement infrastructure to be open-ended with respect to data delivery methods.

3.2.2 Performance instrumentation

There are many techniques that can be used for instrumenting applications, as evidenced by the TAU system which supports techniques ranging from source based instrumentation to virtual machine based instrumentation [SM06].

However, the problem with the plethora of instrumentation methods is that most of them are specific to a particular execution environment. Applications compiled into native code may need to be instrumented using e.g. the DyninstAPI [BH00], while applications compiled to intermediate code such as Java byte code may need to be instrumented using byte code manipulation tools such as ASM [OWC06] or BCEL [ASF06].

The choice of an appropriate instrumentation technique for a particular application is, besides other factors, determined by the level of control over the application. For example, the availability of source code provides a different level of control than just the availability of binaries.

Design-level instrumentation of component-based applications

Our approach targets heterogeneous component-based applications for which source code is assumed to be generally unavailable. This means that instrumentation techniques based on modification of application source code cannot be considered. Instrumenting a heterogeneous application using techniques based on modification of application binaries might require using different methods for different components, depending on the target execution environment of a particular component. However, this conflicts with the requirement for the performance instrumentation technique to be generic.

The only generally applicable techniques for instrumenting component-based applications appear to be based on the same concepts as component-based development itself, i.e. composition and delegation.

The objective is to devise a non-intrusive technique for performance instrumentation of heterogeneous component-based applications that can be used with different component systems and in different execution environments to define design-level performance events related to application execution.

Defining performance events at lower level of abstraction would require using instrumentation techniques specific to the target execution environment, able to

distinguish and instrument low-level application entities. Staying at the design level allows instrumenting an application using a generic technique which employs only the information contained in component-application metadata describing the application architecture.

Low-overhead instrumentation for production-level applications

Even though instrumenting an application to provide performance data is orthogonal to its original function, the instrumentation may influence the application performance, because collecting and storing performance data in response to performance events related to application execution adds certain overhead to the instrumented execution paths. The total overhead imposed by performance data collection on the original application depends on the number of active instrumentation entities generating performance events and the amount of performance data collected in response to performance events.

The requirement on dynamic configuration and control of the measurement infrastructure at runtime serves two purposes. Besides flexibility in association of performance data with performance events, the purpose is to provide control over the overhead of performance data collection, which is especially important in production environment. To reduce the overhead of the entire performance data collection framework, the measurement infrastructure should be able to instruct the instrumentation code to stop emitting performance events at specific locations when its configuration indicates that there are no performance data to be collected in response to events from that location and that the events do not need to be recorded for offline trace analysis.

The objective is to design a performance instrumentation layer that provides runtime control over the execution overhead imposed on an application when a particular instrumentation point is not used for collection of performance data.

3.2.3 Heterogeneous deployment

We aim to enable performance data collection for any off-the-shelf component-based application assembled from components based on different component systems. Since we cannot access the application in earlier stages of its life cycle, we have to integrate the performance instrumentation process with application deployment, which is the last stage in the life cycle before an application is executed. The problem with deployment is that even though the concepts related to deployment of distributed component-based applications are similar or even identical for different component systems, the technical details of the deployment process are typically proprietary and specific to a particular component system runtime. In some cases, the deployment process can be even specific to a particular implementation of a specific component system runtime.

With the advent of the CORBA Component Model [OMG06b], the Object Management Group released a specification on Deployment and Configuration of Distributed Component-Based Applications [OMG06c] which, by providing a common set of concepts and models, attempts to prevent the diversity in vendor and technology specific approaches to application deployment found in existing technologies.

The situation is even more complex with heterogeneous component-based applications targeted in this thesis, because an application may consist of components implemented in different programming languages using different component systems. Even though the OMG specification standardizes most of the concepts and models related to deployment of

component application, the model-driven development approach intended for implementing the specification is not very suitable for creating a deployment runtime supporting heterogeneous component-based applications. Clearly, such applications were not the primary target of the specification, but it does not render the specification useless.

Deployment of heterogeneous component-based applications

The specification currently represents the most comprehensive attempt at standardization of deployment of distributed component-based applications. The deployment process is complex, but we believe that deployment of component-based applications based on many existing component systems can be implemented according to the specification.

The first step in the heterogeneous deployment effort is providing conceptual support for heterogeneous component-based applications, which will enable assembly of components based on different component systems. This exposes the main problem of heterogeneous applications which is related to communication between components from different component systems, because establishing a connection between such components requires bridging incompatibilities between the participating component systems.

Besides supporting heterogeneous component assemblies, the model-driven development approach intended for implementation of a deployment runtime needs to be adapted to allow separation of generic and component-system specific parts of the runtime. Implementing the component-system specific parts of an otherwise generic deployment runtime realizing the deployment process should require considerably less effort than implementing the entire deployment runtime separately for each component system.

The objective is to propose a method for bridging differences between components from different component systems to enable deployment of heterogeneous component-based applications using the deployment process defined by the OMG specification.

Transparent instrumentation during application deployment

An important issue related to component-based applications is that the physical representation of an application (as opposed to an executable image of normal applications) is not easily accessible prior to its launch, which makes instrumenting off-the-shelf component-based applications difficult. Moreover, instrumentation techniques for component-based applications may require modifying application metadata describing the assembly of individual components.

The objective is to make the process of application performance instrumentation transparent by integrating it into the deployment process of heterogeneous component-based applications without the need to expose an application deployer to technical aspects of a performance instrumentation method.

Chapter 4

Measurement Infrastructure

The fundamental concepts related to collecting data for measurement-based performance analysis remain the same, regardless of an execution platform or a component system runtime. Always, there has to be one or more reference quantities which are used to express the performance metrics related to the system under test. These quantities may change independently of the system under test, such as time, in response to its activities, such as number of messages sent to a network, or both, such as the number of packets sent or received by a network interface card. These quantities have to be sampled at moments that are again either independent of the system under test, such as periodic interrupts, or are directly related to the system's activities and represent significant events in its execution. Then the sampled data has to be stored and delivered to a consumer. Optionally, the data can be processed in some way, typically to reduce the amount of data that needs to be stored.

The execution platform, which includes hardware, operating system, and an implementation programming language bring variability into the process, but not on the conceptual level. The hardware and operating system determine the availability of various performance data sources. The programming language determines how the collection of data is implemented and which performance data sources can be accessed natively from a programming language and which have to be accessed some other way. On top of this, a component system (and an application using its primitives) is a concept which is too high-level to influence the process.

Ad-hoc performance measurement solutions targeted at specific applications or middleware platforms often do not differentiate between various aspects of performance data collection. This is especially true for many benchmark applications that exercise and evaluate a system under test. The benchmarks combine implementation code responsible for performance data collection with code responsible for exercising a system under test. Performance events are defined implicitly by code locations which access and store performance data. The association between performance events and performance data is hard-coded in the implementation and the data are accessed either directly, specific to a particular hardware and operating system, or realized using a library for accessing a particular data source. Data storage is typically implemented ad-hoc.

The main benefit of such approach to benchmark construction is often rather low-overhead measurement and apparent low implementation cost, because of the inherently lower complexity of non-generic approaches. On the other hand, the support for performance measurement contained in a benchmark is tied to a particular application or a limited class of applications, and leads to fragmentation of effort that could be used to create a more generic solution.

Our long term goal is to create an infrastructure which will keep the implementations of various aspects of performance data collection clearly separated, providing a framework that could be used for performance measurement regardless of application. The only

difference in using the infrastructure should be related to the responsibility for managing the infrastructure life cycle, and for defining and emitting performance events driving the collection of performance data. The association between performance data and performance events should be defined externally to the source of performance events and performance data should be stored in a generic way.

However, we do not imagine creating a single infrastructure for all possible execution environments, because the environment is the main source of variability that needs to be taken into account in the implementation. The goal is to have an infrastructure for a particular programming language (or a family of similar languages) that could be used with any application in that language.

The design of the measurement infrastructure represents the first step towards the long term goal. A platform independent model of the infrastructure, presented in this chapter, allows constructing a conceptually equivalent implementation of the infrastructure in different programming languages, while adhering to the best development practices associated with a particular language. Additional concepts that may surface during implementation, may be integrated back into the platform independent model if they prove to be generally useful.

In the context of this thesis, the measurement infrastructure is intended to be used with any component-based performance instrumentation suitable for heterogeneous component-based applications, because it does not depend on the specifics of a particular component system. However, when combining components intended for different execution environment, such as native vs. virtual machine, there will need to be an implementation of the measurement infrastructure for use in each of those environments. All implementations will also need to provide a generally accessible interface for external configuration and management, but this can be easily achieved using existing middleware and management platforms, therefore it is not explicitly addressed in this work.

Most of the requirements concerning the measurement infrastructure have been presented in Section 3.1.2. These will be specifically addressed in Sections 4.2 and 4.3. In the following section we describe the main principles in using the measurement infrastructure, identify the main aspects of performance measurement, and introduce a general architecture of the infrastructure capturing those aspects.

4.1 General Design

The general objective in the design of the infrastructure is to split responsibilities related to performance data collection between a user and the infrastructure. A user is expected to specify what data should be collected and where, and the infrastructure is responsible for collecting the requested data. The entire process is envisioned to have the following stages:

1. Initialization of the infrastructure, which includes mainly registration of performance events corresponding to significant events in the execution of the system under test. The events defined during analysis of the system are abstract, and need to be reified within the infrastructure, which will allow associating performance data with the events.
2. Configuration of the infrastructure, which consists of
   a) Association of performance data with performance events registered within the infrastructure. As in case of performance events, the data that need to be collected should be specified during analysis of the

system under test. This stage serves to map the required abstract performance data to specific instances of measurable quantities available in the system.

b) Configuration of the infrastructure for performance measurement, which concerns enabling reception and processing of selected events, reserving storage space for event records with associated data and choosing an appropriate storage policy, and choosing the preferred method of data delivery.

3. Operation, which is driven by the execution of the system under test. The executing system will emit performance events and the infrastructure will collect, process, and store performance data associated with the events.

Depending on a particular application, some of the steps may repeated or interleaved, such as e.g. in case of long running applications, which would register performance events only once during initialization, and then execute regardless of any measurement. For each performance measurement experiment, different performance events will be associated with different performance data.

The initialization stage may differ in various usage contexts, with the difference mainly related to the management of infrastructure life cycle and registration of performance events. In this respect, we primarily target to main use cases.

The first use case corresponds to situations in which a developer uses the infrastructure directly from application code. The code for initializing the infrastructure and registering performance events will have to be written by the developer, which typically results in low number of events with specific semantics. Since the developer has absolute control over application execution, there are no problems with synchronizing the initialization of the infrastructure with the initialization of an application. Even though the infrastructure can be used with any application, this situation is typical for benchmark applications which control the execution of a system under test.

The second use case corresponds to situations in which an application is automatically instrumented by a performance instrumentation tool. The code for initializing the infrastructure and registering performance events will be generated by the tool, which typically results in a large number of generic events, with granularity corresponding to execution of single statements, blocks, methods, or other primitives, depending on the capabilities of an instrumentation tool and resolution inherent to a particular instrumentation technique. In contrast to the previous case, the flow of execution is determined by application code; therefore the instrumentation tool must be able to associate infrastructure initialization with application initialization. This situation is typical for applications that represent the system under test but do not provide the data required for performance analysis. In contrast to benchmarks, these applications may also run in a production environment to allow collection of performance data related to real world usage.

Various aspects of both use cases may be combined, such as a developer using an instrumentation tool to define and register only a few specific performance events in the application source code, but that does not make much difference in the initialization stage, even though the developer may make it easier for the instrumentation tool to integrate infrastructure initialization with application code.

4.1.1 Architecture overview

The main objective in the design of the measurement infrastructure is to maintain clear separation between various aspects of performance data collection. On the highest level of abstraction, this separation can be illustrated on the architecture of the TAU performance system [SM06] (see Section 2.3 for an overview), which consists of three layers responsible for instrumentation, measurement, and analysis.

Addressing neither performance instrumentation nor performance data analysis, the measurement infrastructure corresponds to the middle layer of TAU architecture and is only responsible for performance measurement. This should allow using the measurement infrastructure both by developers in benchmark applications, and by various performance instrumentation tools.

On the level of abstraction corresponding to performance measurement, we differentiate between two groups of abstractions. The first group corresponds to activities directly related to performance measurement that are typically driven by the execution of an application sharing the address space with the measurement infrastructure. These aspects performance event sources which drive the measurement, performance data which provide the values of various measurable quantities, event processing which associates performance events with performance data, and data storage which allows keeping event profiles or traces with associated data.

The other group of abstractions corresponds to activities that are typically driven by external processes, not directly related to the execution of the application under test. These include data delivery which provides external access to the stored performance data, and management which coordinates the activity of the entire infrastructure.

The above abstractions should be clearly separated to ease maintenance, lower the barrier for extending the infrastructure, and simplify potential reuse of selected infrastructure parts. The internal architecture of the TAU measurement layer contains similar abstractions, but encapsulates event processing, data storage, and data delivery into a common block providing separate support for profile-based and trace-based measurement modes.

The proposed general architecture of the measurement architecture is depicted in Figure 1. The components in the diagram correspond to different subsystems of the infrastructure realizing the above abstractions and the connections capture interactions between the subsystems. The gray block on the right side of the diagram corresponds to the main processing “pipeline” of the infrastructure, i.e. the part which is driven by performance events emitted during execution of the application hosting the infrastructure. The subsystems on the left side of the diagram correspond to the part of the infrastructure which is driven by external activity.

Application-driven subsystems

The application driven part of the infrastructure executes in the context of the application and includes the *Event Sources*, *Performance Data*, *Event Processing*, and *Data Storage* subsystems.

Event Sources

The runtime entities of the *Event Sources* subsystem are responsible for emitting performance events that drive the collection of performance data. Each event source may emit multiple performance events. Depending on the usage context, event sources may be

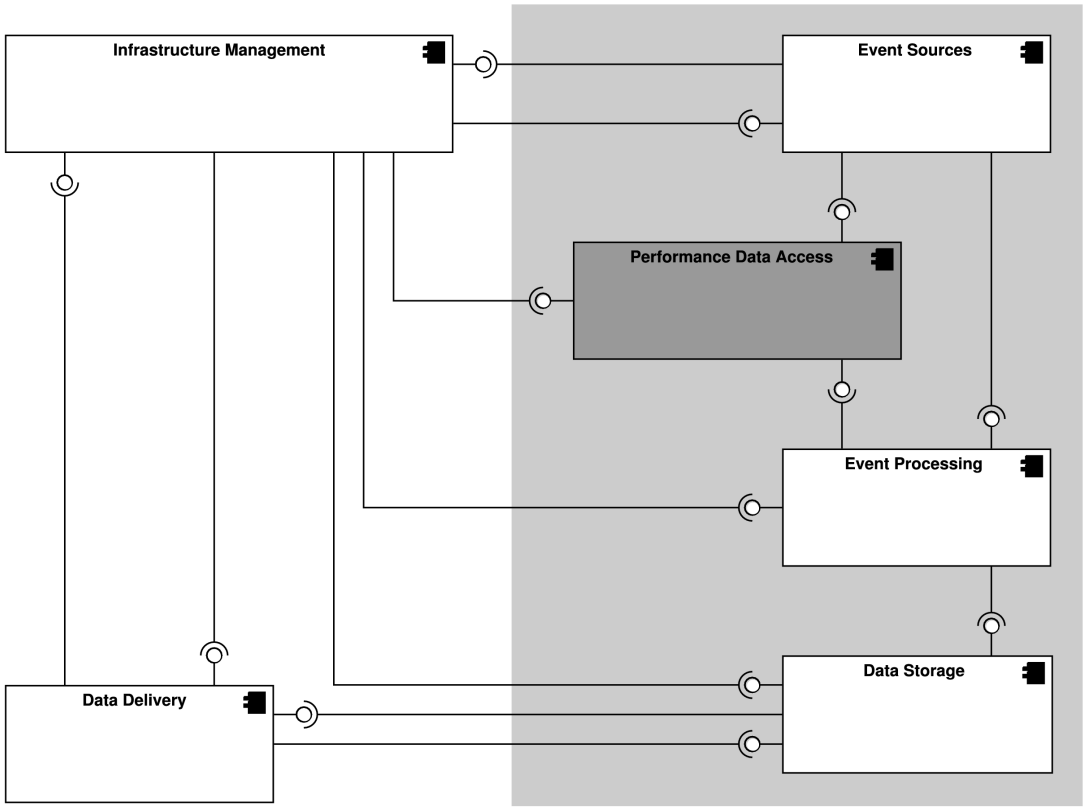


Figure 1: Overview of the measurement infrastructure architecture

implemented by the measurement infrastructure, by performance instrumentation code, or by an application developer using the infrastructure. The subsystem interacts with tree other subsystems using one provided and three required interfaces.

The interaction with the *Infrastructure Management* subsystem is two-stage. Since the event sources need not be implemented by the infrastructure, they may be created independently, but have to register with the infrastructure. During registration, event sources provide the measurement infrastructure with a management interface, which is used both to query information about supported events and to enable or disable the event source or any of its performance events. The interface is also used to provide the event sources with references to entities from the *Performance Data* and *Event Processing* subsystems.

With respect to performance data collection, the *Performance Data* subsystem is used to include precise timing information in events emitted by the event source, while *Event Processing* subsystem is used for reporting events to the measurement infrastructure. If a particular event source is disabled, no events are reported.

Performance Data

The *Performance Data* subsystem is responsible for providing a unified interface and generic access to different sources of performance data. The performance data typically have the form of counters or gauges and provide readings of various quantities that can be observed in a running system. The subsystem interacts with three other subsystems using three provided interfaces.

The interaction with the *Infrastructure Management* subsystem concerns creation of measurement contexts capturing a selection of performance data associated with a particular performance event. The measurement contexts are then used by the *Event Processing* subsystem to initiate collection of performance data and to access the data.

Besides that, the *Infrastructure Management* subsystem also obtains a reference to a time source, which it provides to event sources. This is necessary to keep all timing information associated with performance data consistent.

The *Performance Data* subsystem is also a pure service subsystem, which only defines provided interfaces. This allows using the subsystem in other applications and in fact measurement infrastructure is basically built around this subsystem. This will be discussed later in more detail.

Event Processing

The entities of the *Event Processing* subsystem are responsible for processing performance events emitted by event sources. Processing events means collecting performance data associated with a particular event and creating a data record containing event-specific information along with the performance data, which is then stored in memory. The *Event Processing* subsystem interacts with three other subsystems using two provided and two required interfaces.

Through one of the provided interfaces, the *Event Processing* subsystem is used by the *Event Sources* subsystem. The other provided interface is used by *Infrastructure Management* to provide references to entities from the *Performance Data* and *Data Storage* subsystems.

The *Performance Data* subsystem is used to collect performance data associated with a particular performance event. The association is maintained by the *Infrastructure Management* subsystem, and is part of the provided reference to a *Performance Data* entity.

The *Data Storage* subsystem is used to store event records along with performance data as well as aggregate statistics for specific quantities contained in the performance data. As with the *Performance Data* subsystem, a reference to a configured *Data Storage* subsystem entity is provided by *Infrastructure Management*.

Data Storage

The *Data Storage* subsystem is the last stage of the entire event processing pipeline and is responsible for storing performance data in memory to avoid delaying the execution of the application. Performance data associated with a particular event type can be either in form of event records, which capture all occurrences of an event, or in form of aggregates associated with a particular quantity contained in an event record, updated on each occurrence of an event. The subsystem interacts with three other subsystems using three provided and one required interface.

Through one of the provided interfaces, the *Data Storage* subsystem is used by the *Event Processing* subsystem. The other provided interface is used by *Infrastructure Management* to create and configure Data Storage entities provided to *Event Processing* entities. The interface is also used to configure a storage policy and to provide a reference to *Data Delivery* entities in case they need to be notified about exhausted storage space.

The third provided interface is used by *Data Storage* entities to query and access the available performance data and to request notifications when e.g. the available storage capacity drops below a defined limit.

Externally-driven subsystems

The part of the infrastructure driven by external activities executes in a different context than the processing pipeline and includes the *Data Delivery*, and *Infrastructure Management* subsystems.

Data Delivery

The *Data Delivery* subsystem is responsible for delivering the collected performance data to the consumers who initiated the process. The method of delivery may include writing event trace and profile traces to disk or a database, sending data over the network directly to consumers, or to a monitoring console, etc. Depending on the delivery method, the *Data Delivery* subsystem may be either active and poll the *Data Storage* subsystem for new data, or passive and only react to external requests for data or notifications from the *Data Storage* subsystem. Active delivery subsystem will need to execute in its own context. The subsystem interacts with two other subsystems using two provided and two required interfaces.

One of the provided interfaces is used to provide *Data Storage* entities with a reference for notifying the *Data Delivery* entities when their storage capacity drops below a configured limit. The required interface to the *Data Storage* subsystem allows the *Data Delivery* entities to query the available storage space and access the stored performance data.

The other provided interface is used by *Infrastructure Management* to configure data delivery methods and to associate *Data Delivery* entities with *Data Storage* entities. The required interface to *Infrastructure Management* may be used to query configuration information by active *Data Delivery* entities.

Infrastructure Management

The *Infrastructure Management* subsystem represents the key part of the measurement infrastructure responsible for configuration and coordination of all other subsystems. It also maintains an externally accessible management model of the application created from the names of event sources and supported performance events, which allows configuring a measurement experiment from outside. The subsystem interacts with all other subsystems using five required and two provided interfaces.

One of the provided interfaces serves for registration of *Event Sources* entities, the other serves to provide the *Data Delivery* entities with access to the management model of the application.

The required interfaces are used to create and configure subsystem entities and to provide them to entities in other subsystems. In case of *Event Sources* entities, the interface is also used to query information about supported events and to configure which events should be reported to the infrastructure.

4.1.2 Infrastructure realization

The measurement infrastructure will be realized as a library which will share the address space of an application providing performance events that drive performance data

collection. The library will provide an interface that will be used either by an application itself or by instrumentation code integrated with an application to initialize the infrastructure, register performance event types, and provide performance events to the infrastructure. In case of distributed applications, the infrastructure will be present in each address space occupied by an application.

This is basically the only feasible physical model of using the infrastructure with applications spanning multiple address spaces, even on a single node. Since the infrastructure has to exert minimal influence (mainly in terms of delays) on the execution of the application providing performance events, the subsystems that process performance events in the application execution context must be in the unconditionally in same address space, even though it would be technically possible for them to be separated from the application.

Since the other part of the infrastructure executes in a different context, it may be tempting to separate the two parts of the infrastructure by leaving the event processing in application address space and creating a single Infrastructure Management and Data Delivery subsystem per node. Even though the two parts of the infrastructure execute in different contexts, they are rather tightly coupled, with the Infrastructure Management subsystem responsible for life cycle of entities from other subsystems. While separating the two subsystem groups would be technically possible, there are no benefits to it and it would only immensely increase the complexity of using the infrastructure.

To manage an experiment in context of a distributed application, an additional management layer has to be created, which will aggregate the configuration and management of all instances of the measurement infrastructure. This is conceptually similar to aggregating various sources of management data provided through SNMP or JMX, therefore we will not address this aspects of using the infrastructure in distributed environment.

Infrastructure implementation

The platform independent model of the measurement infrastructure allows creating an implementation of the infrastructure for use from within a particular programming language. However, most implementations will not be pure, using only a single programming language, because of the need to access various data sources through the Performance Data subsystem. While pure implementations are possible and sometimes desirable, restricting the infrastructure implementation to a single language will also prevent using data sources that have to be accessed using an API not available in that language. This is especially true for higher level, virtual machine-based languages such as Java or C#.

Compared to other subsystems, the Performance Data subsystem is specific in that it will have to use modules written in other, non-native, languages to provide access to various performance data sources.

We have already mentioned that the Performance Data subsystem is somewhat special in being a pure service subsystem, with the rest of the measurement infrastructure built around, or more precisely, on top of it. This basically determines the physical architecture of the measurement infrastructure, which consists of two layers, as illustrated in Figure 2. The top layer contains native subsystem parts, implemented in the target programming language of the infrastructure, while the bottom layer contains non-native subsystem parts, implemented in different programming languages.

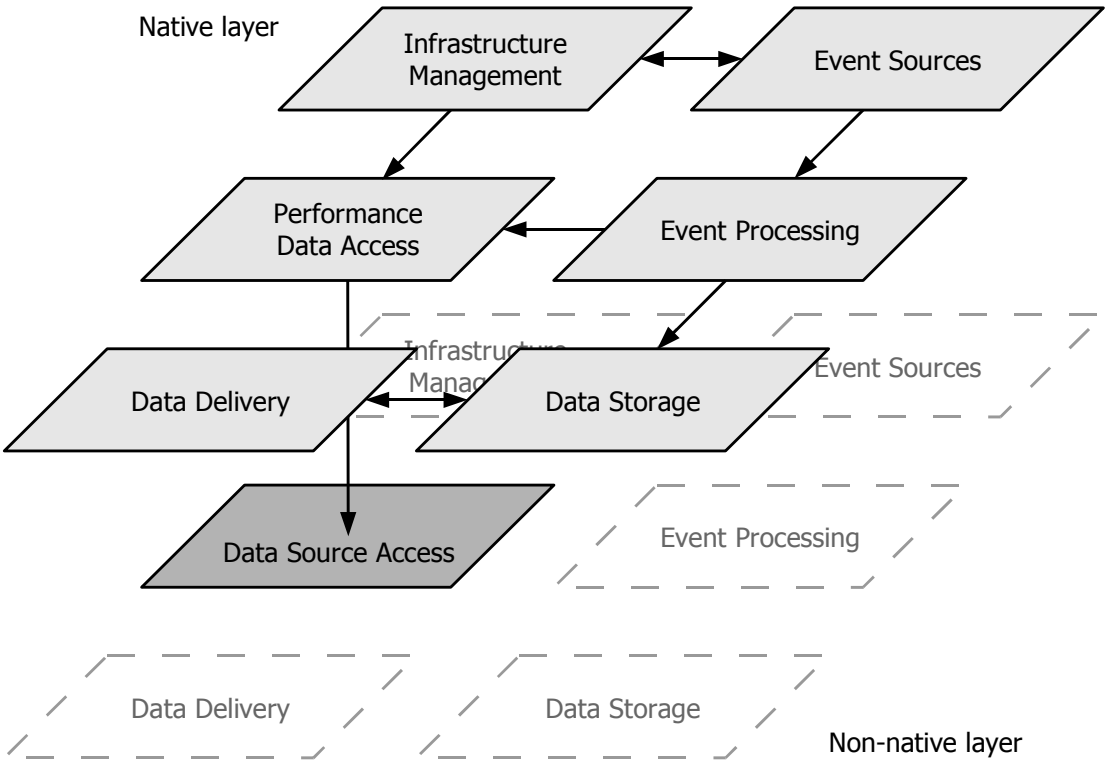


Figure 2: Measurement infrastructure layers

In Figure 2, only the Performance Data subsystem has a non-native part, which consists of modules for accessing various data sources. The native part of the subsystem uses these modules through language bindings and provides a unified interface for accessing all performance data provided by individual data sources.

Since the Performance Data subsystem does not depend on any other part of the measurement infrastructure, the entire subsystem can be isolated to a standalone project, which can be developed independently of the measurement infrastructure. Considering the subsystem alone, its physical architecture will consist of three layers, as illustrated in Figure 3. The first layer will contain implementation of the Performance Data subsystem in various programming languages, the middle layer will contain language bindings for accessing various performance data sources, and the bottom layer represents the individual data sources. The motivation behind the separation is to allow using the subsystem by other applications, which may need to access a wide range of performance data but do not need the measurement infrastructure. This may provide incentive for extending the subsystem with support for additional data sources.

On the other hand, an implementation of the measurement infrastructure for use with a particular programming language will only have to focus in creating native implementation of other infrastructure subsystems around the Performance Data Subsystem which may be already available for that language.

This does not imply that the measurement infrastructure must be implemented natively for each target programming language. The layout of physical architecture of the infrastructure shown in Figure 2 contains shadow images of all subsystems also in the non-native layer. This means that other subsystems of the infrastructure may be also implemented in a non-native language made available to the rest of the infrastructure through language bindings. This approach may be used e.g. to provide a scripting language

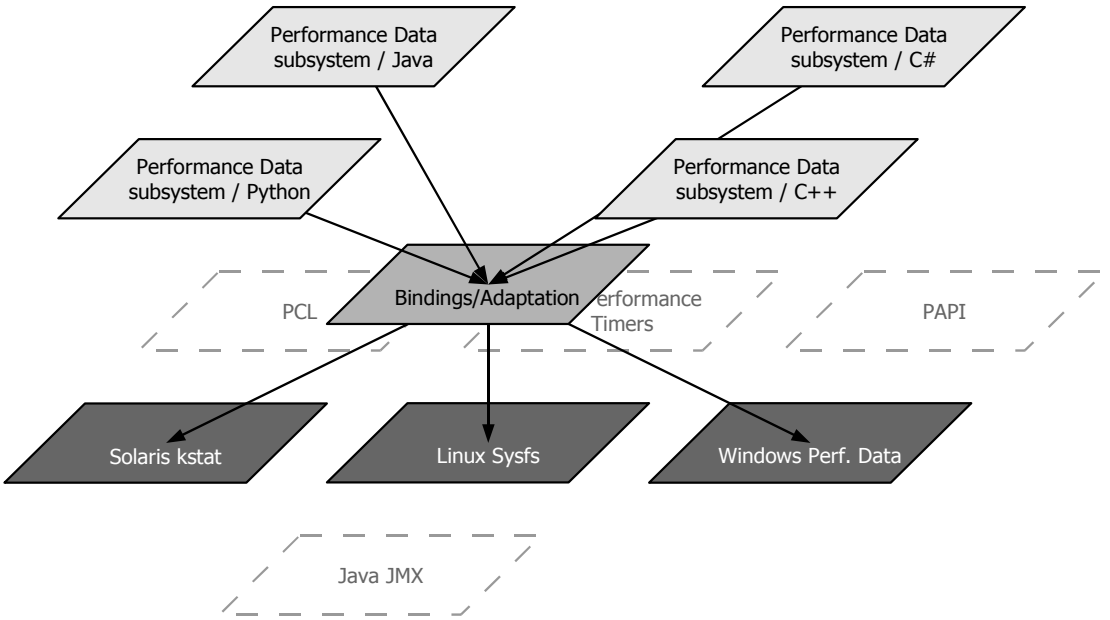


Figure 3: Performance Data subsystem layers

with a high-performance Event Processing and Data Storage subsystems, or to provide a low-level language with an implementation of Data Delivery subsystem written in higher level language better suitable for middleware-based communication.

Since the Performance Data subsystem is useful even without the rest of the measurement infrastructure, the following section deals with the design of the Performance Data subsystem, separately from the rest of the infrastructure.

4.2 Performance Data Subsystem

The role of Performance Data subsystem in the measurement infrastructure is to provide access to different sources of performance data and timing information. Performance data are provided by various parts of a computer system, both hardware and software. The data typically have the form of counters or gauges related to occurrence of external and internal events, resource availability and consumption, etc.

Performance data from various sources can be usually accessed through a software layer, either an operating system or a library, which defines an API that enables obtaining the values of particular counters or gauges. This requires all performance data values accessible through the API to be uniquely identified. However, there are many performance data sources which have to be accessed through different software layers, each providing a different API to access the performance data values, and each using a different naming scheme to identify the data. To enable accessing performance data from various sources, the Performance Data subsystem has to define a uniform naming scheme for identifying performance data, and a generic method for obtaining selected performance data values.

These requirements are reflected in the general architecture of the Performance Data subsystem depicted in Figure 4. On the top level of abstraction, there are multiple sources of performance data and timing information to which the subsystem provides unified access. Each *Data Source* provides a specific API for obtaining the values of performance counters and gauges it exposes. Each *Time Source* is basically a performance data source

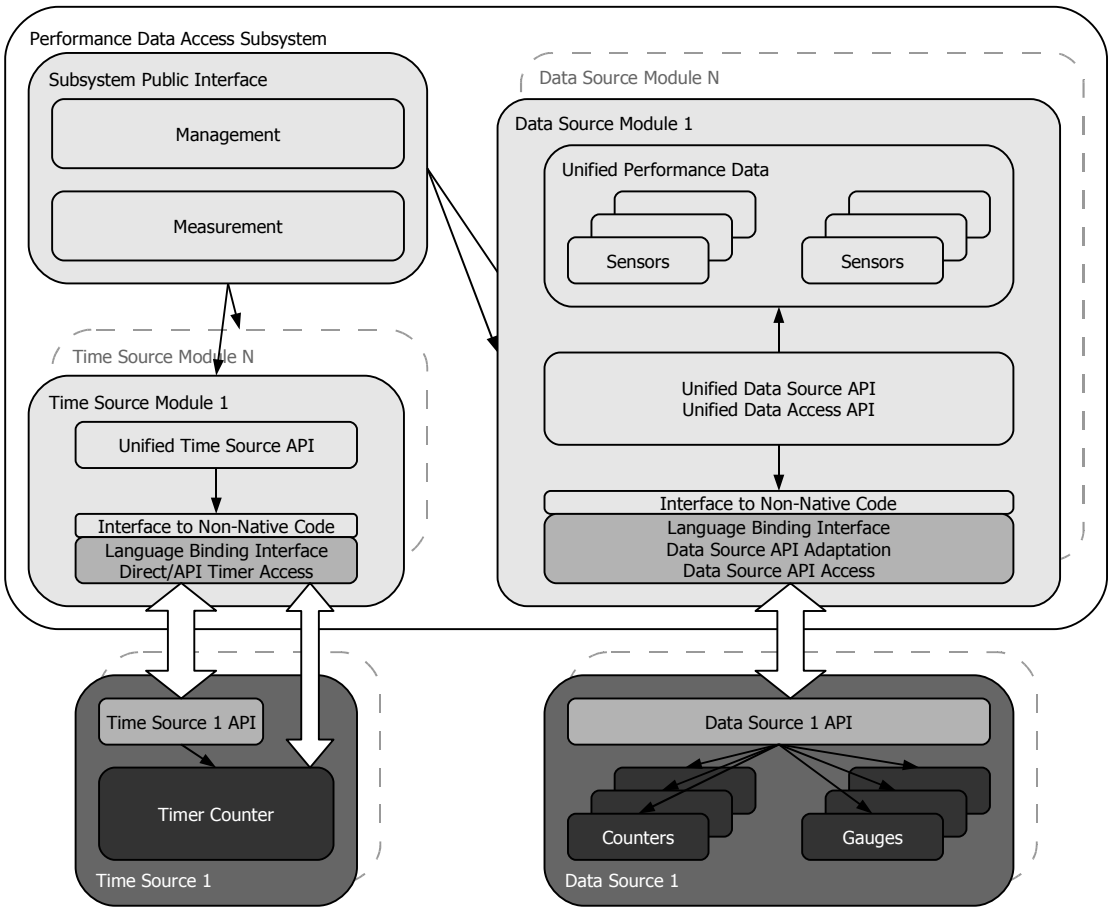


Figure 4: General architecture of the Performance Data subsystem

with the difference that it provides only a single counter which is identified implicitly and provides a single type of values; therefore the API for accessing it is usually simpler.

The subsystem itself contains a *Time Source Module* and a *Data Source Module* for each source of timing and performance data it supports, and a *Subsystem Public Interface* which provides information about available performance data as well as unified access to their values.

Since the *Subsystem Public Interface* represents the visible part of the Performance Data subsystem, it defines unified interfaces for accessing performance and timing data. A *Time Source Module* has to provide a unified timer interface, and a *Data Source Module* has to do the same for performance data. However, this is more complex because each *Data Source Module* provides access to multiple performance counters and gauges, each having a different data type and semantics.

The subsystem therefore defines a Sensor as common abstraction over different kinds of data provided by performance data sources, which allows querying and accessing performance data in a unified manner. A sensor is associated with metadata describing its type and semantics, as well as with a *Data Source Module* responsible for obtaining its values. The mapping between sensors provided by the subsystem and the counters and gauges provided by a particular performance *Data Source* is the responsibility of the respective *Data Source Module*. The *Management* part of the *Subsystem Public Interface* allows querying available sensors and time sources as well as obtaining sensor metadata

describing the associated performance counters and gauges. The *Measurement* part allows choosing a set of sensors for repeated sampling of their values.

As shown in Figure 3, an implementation of a Performance Data subsystem is always targeted for use in a specific programming language and has to support modules written in a different, non-native, programming language providing access to performance data sources using an API which is not available in the native language. This is the responsibility of the *Time Source* and *Data Source* modules in the subsystem architecture.

Compared to a *Data Source Module*, a *Time Source Module* is much simpler because it does not have to deal with identification of multiple counters and different data types. Each time source provides a unified time source interface defined by the subsystem, which can be used to obtain the value of the underlying timer counter. If a particular time source can be accessed natively from the subsystem target programming language, the respective *Time Source Module* must only ensure adaptation to provide a unified interface. However, high-precision time sources are typically platform specific and need to be accessed using specialized interfaces or directly, e.g. by reading a processor's time stamp counter by using a processor specific instruction. Even though direct access is possible, timing sources are better accessed using an intermediate library, e.g. to avoid dealing with specifics of different processors.

Since this may not be possible in all programming languages, a *Time Source Module* may use code written in different programming language to access a particular time source. Besides accessing the time source, this code is responsible for providing an interface which is used by the execution platform of the subsystem target language to integrate the non-native code with native code. This language binding interface is specific to a particular platform and needs to have a native counterpart to which the environment binds the non-native code.

A *Data Source Module* is typically more complex, because it has to provide uniform interface for accessing performance data, which is conceptually similar to a unified time source interface, but needs to support different data types and multiple values. In addition, the module also has to provide the subsystem with unified data source interface, through which the *Management* part of the *Subsystem Public Interface* obtains information about available performance data and their type. Similarly to the *Time Source Module*, if a data source can be accessed natively from the subsystem target programming language, the respective *Data Source Module* must only provide mapping between sensors and the available performance data, and adaptation between the unified data access interface needed for measurement and the interface used to access the data source. However, as with different time sources, the most useful sources of performance data are typically available only through a system or library level interface that is not available for higher-level programming languages.

Again similar to the *Time Source Module*, a *Data Source Module* may use code written in different programming language to access a particular data source. In most cases, the adaptation between the unified data access interface and the data source interface will be performed within the module, using direct language bindings to the non-native interface. However, in some cases the data source interface may be too alien or unsuitable for direct binding to the native language of the Performance Data subsystem, and will need to be adapted to the unified data access interface within the non-native code, which in turn may need to keep some internal state to provide the adaptation.

4.2.1 External interfaces

From the point of view of the measurement infrastructure (see Figure 1), the most important part of the Performance Data subsystem is the public interface provided to the infrastructure. This interface has to allow the infrastructure (in response to external requests) to query the available performance data, associate them with performance events, and collect and store the performance data values in response to performance events occurring during application execution.

The Performance Data subsystem depicted in Figure 4 has a Management part, responsible for providing information, and a Measurement part, which is responsible for providing access to performance and timing data. However, the Measurement part of the subsystem can be only accessed through the Management part.

The subsystem uses sensors as an abstraction for various kinds of performance counters and gauges and its Management part provides information about available sensors. All sensors must uniquely identify a particular counter or gauge provided by any supported performance Data Source Module. The subsystem therefore has to use a unified naming scheme to identify sensors and the scheme must be able to contain the naming schemes of all supported data sources.

In case of time sources, this is not necessary because time sources are associated with only a single timer counter, and the Management part of the subsystem only provides information about available time sources and their properties.

Sensor identification

Performance counters or gauges are always associated with some logical or physical part of a computer system. Processor performance counters are associated with system processor; in case of multiple processors there are multiple sets of performance counters associated with each processor. Network packet counters may be associated with the system as a whole, but also with individual network interface cards. The amount of allocated memory may be associated with individual processes, but also with the entire system.

Most data sources provide multiple counters and gauges associated with one or more parts of the system, and these parts, their instances in the system, and the semantics of individual counters and gauges determine the naming scheme used to identify them.

Considering the PAPI and PCL libraries for accessing processor performance counters, they use a flat naming scheme, with different counters identified by a symbolic constant defined in library header files. To access the counters, a user has to specify a list of counter identifiers and the library ensures their activation on a particular processor and provides counter readings. Grouping of similar counters is performed using a naming convention for deriving symbolic counter names.

Considering the Linux operating system, the easiest way to obtain performance data is by reading files from the *proc* and *sysfs* virtual file systems. The identification of performance counters and gauges therefore corresponds to locating a file in a file system. However, the naming is very diverse. In some cases a directory structure is used to separate different subsystems, their parts, and individual counters and gauges, in other cases a single file may contain multiple counters, some of them for multiple logical entities, and some of them for the entire system.

The *Windows Performance Data* subsystem of the Windows operating system provides named performance objects which correspond to various logical and physical parts of the system. Counters and gauges are always associated with a particular performance object. In some cases, there may be multiple named instances of performance objects, such as when there are multiple processors, network interface cards, or user processes. The naming is therefore hierarchical, with the name of performance object at the first level, name of a performance counter at the second level, and name of a performance object instance at the third level.

The kstat subsystem for accessing performance data in the Solaris operating system uses a similar structured naming scheme, and consists of class, module, and instance to identify an entity providing named values corresponding to counters and gauges.

Both Windows and Solaris operating systems use similar concepts for identifying performance data. First they identify an entity (and optionally its instance) providing performance data and then identify a particular counter or gauge. This scheme should be suitable for most, if not all, performance data sources. Since the identification of an entity may have multiple components, such as in case of the Solaris operating system, and basically serves to group related performance data together.

However, since the Performance Data subsystem cannot realistically define the naming for all entities and performance data from different data sources, each Data Source Module in the subsystem is responsible for creating sensor names and for mapping those names to performance data in the original data source. To avoid clashes in the sensor name space, each Data Source Module is provided with its own sensor name space, which adds an additional component to sensor name, which therefore consists of four parts:

1. identifier of a Data Source Module,
2. identifier of a sensor group, which corresponds to identification of an entity providing a group of related sensors,
3. identifier of a sensor within a group, which corresponds to a particular counter or gauge provided by the entity, and
4. identifier of a sensor instance, which is optional and corresponds to a particular instance of the entity providing the sensors.

This structure of sensor name can be conveniently represented as an URL, which allows using a simple string data type in the communication with the Performance Data subsystem. The following two fragments show the structure of the URL encoded sensor name and several examples.

```
[sensor://]<datasource>/<group>/<sensor>[#<instance>]
```

```
linux.procfs/system.stat/cpu.user
linux.procfs/system.stat/cpu.iowait#0
linux.procfs/task.statm/resident#self

win32.perfdata/memory/pool.paged.bytes
win32.perfdata/process/working.set#345

solaris.kstat/net.hme/obytes#hme0
solaris.kstat/cpu.sysinfo/kernel
```

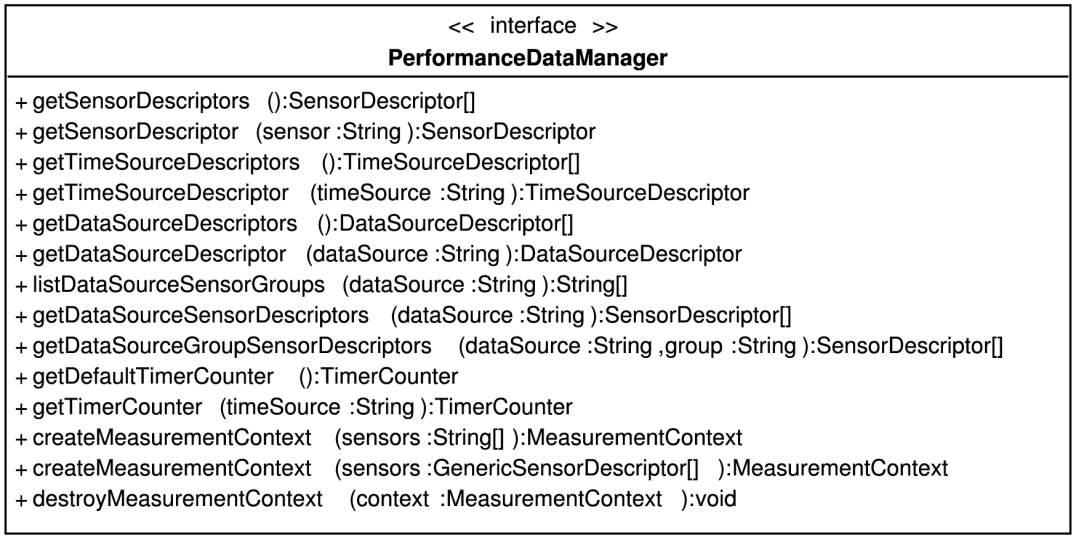


Figure 5: External management interface of Performance Data subsystem

While this naming scheme allows accessing various data sources, the selection of sensors is not portable between different operating systems. However, this can be solved by creating a virtual Data Source Module, which will use the sensors provided by other modules and define a common naming system for selected performance data sources. Since the instance names of entities providing performance data can be arbitrary and cannot be easily standardized, such as e.g. the names of network interfaces, a virtual Data Source Module may provide two different sensor groups for the same performance data entity, one providing native instance names and the other numerical identifiers corresponding to the ordering of instances exposed by a particular data source.

```
common/system.processor/userTime
common/network.interface/packetsSent#0
```

Management interface

The Management part of the Performance Data subsystem can be accessed through a PerformanceDataManager interface shown in Figure 5. The interface provides methods to query available sensors, time sources, and data sources, which return descriptors that allow querying information specific to a particular entity. Most methods serve for obtaining information, with the exception of getTimeSource and createMeasurement-Context, which provide access to the Measurement part of the subsystem.

The structure of descriptors returned by the informational methods is shown in Figure 6. Through a generic ancestor represented by GenericDescriptor, each descriptor provides basic information such as identifier of a sensor, time source, or a data source within the subsystem, along with a human readable name and description. TimeSourceDescriptor provides information about the resolution and precision of a time source. Sensor-Descriptor provides information about sensor type, i.e. whether it is a counter or a gauge, the type of the value it provides, and about sensor instances. SensorInstanceDescriptor shares a common ancestor with a sensor descriptor and only allows to get the descriptor of a sensor to which it belongs.

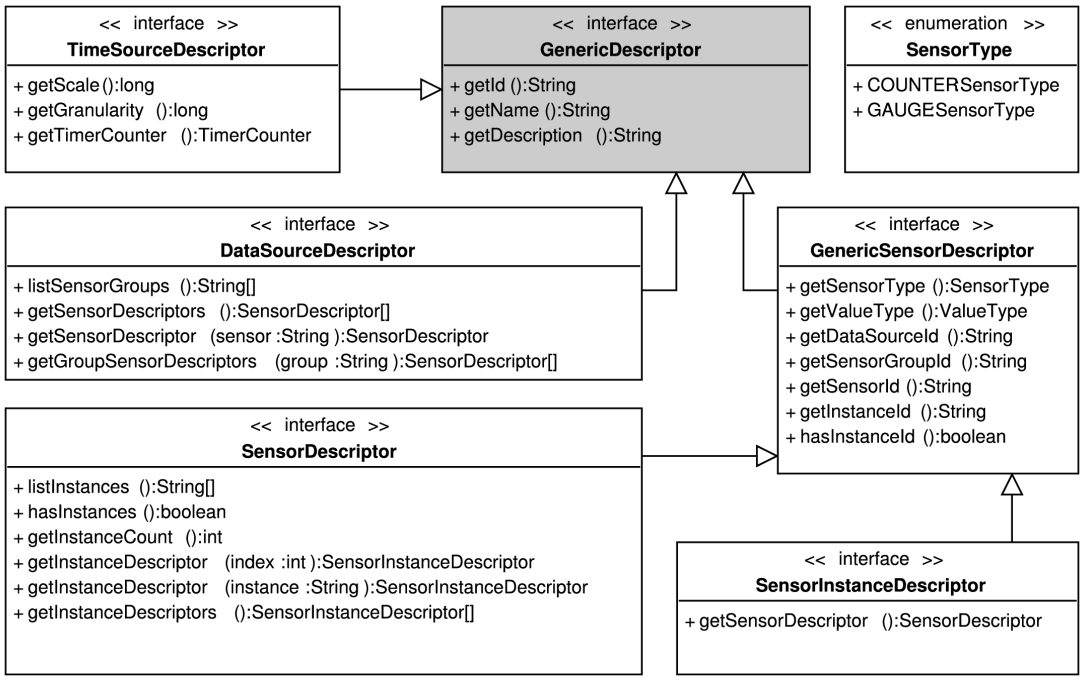


Figure 6: Structure of Performance Data subsystem entity descriptors

Measurement interface

The Measurement part of the Performance Data subsystem can be accessed through the MeasurementContext and TimeSource interfaces shown in Figure 7. These interfaces can only be obtained via the PerformanceDataManager interface. The TimeSource interface can be obtained by calling the getTimeSource method, which has an identifier of a time source as a parameter. This allows using a particular time source if multiple time sources are available. The TimeSource interface is very simple in that it provides only a single type of data, and besides reading the value of an underlying timer counter using the getTime method, it allows determining the resolution of the time source using the getResolution method, and the precision of the time source within the resolution using the getPrecision method. A time source is expected to allow concurrent usage by multiple threads.

The MeasurementContext interface can be obtained by calling the createMeasurement-Context method, with a list of sensors that will be accessed through that measurement context. When creating an instance of a MeasurementContext, the subsystem prepares its internal data structures for accessing the performance data corresponding to the selected sensors. This may require allocating temporary buffers required for obtaining the desired data from the data sources.

To use a MeasurementContext to obtain performance data, the prepare method must be called first, during which the internal implementation of the subsystem prepares for sampling the values of performance counters and gauges in the data sources. This may include e.g. seeking to the beginning of file in case of virtual file systems such as *proc* and *sysfs* in the Linux operating system.

To sample the values of performance counters, the sample method must be called on the MeasurementContext interface. This initiates access to performance data in various data sources corresponding to the selected sensors. Since there may be several sensors to sample, each Data Source Module must perform only necessary operations to take a

snapshot of the data, which can be related to a particular time. No parsing of the received data should be done to allow taking a snapshot of the data as fast as possible. The SamplingResult returned by the method informs the caller about the outcome of the operation. Besides SUCCESS and FAILURE, a special result ANOMALY may be returned, which indicates that anomalous activity occurred during sampling. A Data Source Module responsible for sampling the value of a particular sensor would return this status e.g. when it had to allocate or reallocate memory during sampling, or if it had to use more system calls then usual to take a snapshot of the data. When analyzing performance data, values marked as anomalous may be assigned different weight or completely discarded.

To decode the data snapshots into sensor values that can be accessed by a user, the decode method has to be called. This instructs the subsystem, and in turn each Data Source Module, to parse and decode the contents of internal buffers holding a snapshot of the performance data values. In some cases, the snapshot data only need to be copied, but in general the data may need to be converted e.g. from ASCII representation to binary values, such as in case of values read from virtual files on *proc* and *sysfs* file systems, or the data are encapsulated in complex data structures that need to be traversed, such as in case of performance objects from Windows Performance Data subsystem.

After decoding the data, the values of individual sensors may be accessed through a generic data interface. In a single threaded environment, the prepare, sample, and decode methods will be typically called in sequence, without the need to separate them. A convenience update method, which does exactly that, has been provided for such cases. However, in multi-threaded environment, the calls to these methods may be separated. Especially the decode method and subsequent access to sensor values may be performed in a different thread, which allows decoupling sampling and storage of performance data.

However, a single instance of a MeasurementContext can be safely used only by a single thread at a time and is expected to be generally thread unsafe. This is to avoid locking in situations where a user of the subsystem has full control over the number of threads that may be using a particular MeasurementContext instance. However, when decoupling sampling and storage into different threads, the access to the MeasurementContext instance must be synchronized to avoid race conditions. Since storing the data may take some time during which another request to sample data may come, the thread initiating the sampling may use multiple instances of MeasurementContext to avoid waiting for the storage thread to finish its work.

An instance of a MeasurementContext can be duplicated using the clone method, which is conceptually similar to calling a createMeasurementContext method on the PerformanceDataManager interface, with the same arguments that were using to create the initial instance. However, the internal implementation of the Performance Data subsystem and in turn each Data Source Module responsible for obtaining the values of selected sensors may take different steps to conserve resources and still provide multiple MeasurementContext instances.

Since calling the clone method may initiate memory allocations, it should not be done in the context responsible for initiating sampling of data. The envisioned usage in multi-threaded environment is creating a pool of MeasurementContext instances that would be shared by the sampling and storage threads. The thread responsible for sampling would pick an available MeasurementContext instance, use it to sample performance data, and submit it to the storage thread. The storage thread, when finished storing data, would return the instance back to the pool.

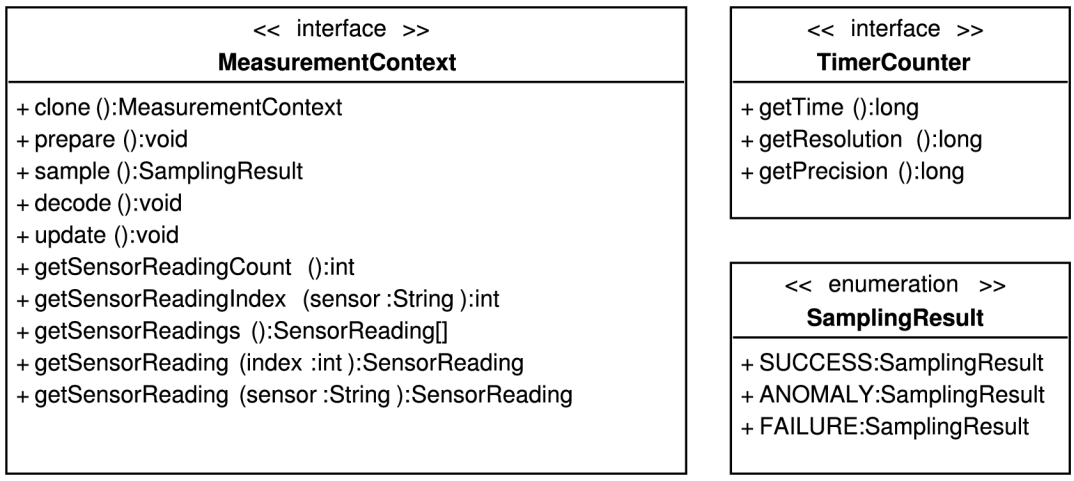


Figure 7: Performance Data subsystem measurement interface

This is a key concept that allows using the Performance Data subsystem within a Performance Measurement Infrastructure (see Figure 1 for architecture overview). For each performance event type, a MeasurementContext instance (or a pool of instances) configured to obtain the values of performance data associated with a particular performance event will be created by the Infrastructure Management subsystem. These instances will be provided to the Event Processing subsystem, which will use them to sample performance data in response to events emitted by the Event Sources subsystem. After sampling, the MeasurementContext instances will be submitted to the Data Storage subsystem which will store the contained data.

When not used anymore, each MeasurementContext instance should be destroyed by calling the destroyMeasurementContext method on the PerformanceDataManager interface. This allows the data sources to release resources allocated for accessing the performance data. This method is required even in execution environments with garbage collection, because other resources like file descriptors are allocated during creating of a measurement context. Specifically, some resources may be also allocated by non-native parts of Data Source Modules, which need to be manually released to avoid resource leaks.

Sensor readings

Aside from internal buffers used to take a snapshot of performance data, each MeasurementContext is assigned storage space for values of all sensors selected for sampling, including all their instances unless specific instance have been selected. The storage space is used to store sensor values obtained during parsing and decoding the contents of internal snapshot buffers after calling the decode method.

The decoded sensor values are accessed through a SensorReading interface depicted in Figure 8, which is provided for each sensor selected for sampling. The main purpose of the interface is to provide generic access to the data associated with a MeasurementContext instance. Apart from the actual sensor values, the SensorReading interface also provides description of the data. This allows storing the values of sensors associated with a MeasurementContext in a generic way, without having to define a fixed data structure beforehand, which is especially important in generic environments such as the Measurement Infrastructure.

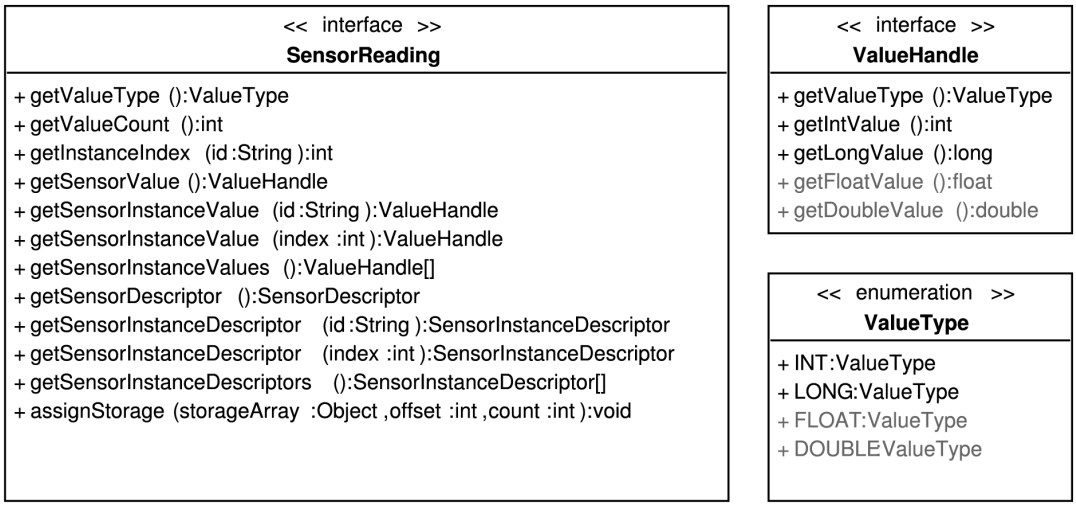


Figure 8: Interfaces for accessing MeasurementContext sensor values

A SensorReading interface for each sensor associated with a MeasurementContext can be obtained by calling the appropriate methods, such as e.g. the getSensorReadings method. Each SensorReading provides access to values associated with a single sensor, including all (or the selected) instances. To access a particular value, a ValueHandle interface must be obtained for that value and a method corresponding to the value data type must be called. This indirection is required to provide generic access to the data and also to allow the data source modules to efficiently allocate memory for sensor values.

Since SensorReading instances are created along with a MeasurementContext instances, the value handles can be obtained before any data have been collected. The SensorReading instances therefore need to be traversed only once and not after every call to the decode method, as illustrated in Figure 9. After a user entity obtains a MeasurementContext instance, it calls the getSensorReadings method to obtain SensorReading instances for all sensors in the measurement context. For each SensorReading instance it determines the type of values it provides, obtains descriptors of sensor instances and obtains all ValueHandle instances for all values described by a particular SensorReading. Then it prepares its own internal structures for subsequent access to sensor values, which is represented by the buildDataDescription method. After preparing its internal structures, the user entity proceeds with measurement by calling the prepare, sample, and decode methods in a closely unspecified sequence. In each iteration, after calling the decode method, the user entity uses its internal structures to obtain sensor values by calls to all ValueHandle instances associated with the MeasurementContext instance.

The concept of accessing sensor value storage through value handles can be also used internally by Data Source Modules, which may somewhat slow down decoding due to additional indirection, but on the other hand it allows changing the underlying sensor value storage.

In the context of the Performance Measurement Infrastructure, this can be exploited by entities from the Data Storage subsystem, which may assign their own storage buffers to sensor readings to avoid copying the decoded sensor values. After swapping the storage buffers, the MeasurementContext instance may be reused for next sample by entities from the Event Processing subsystem.

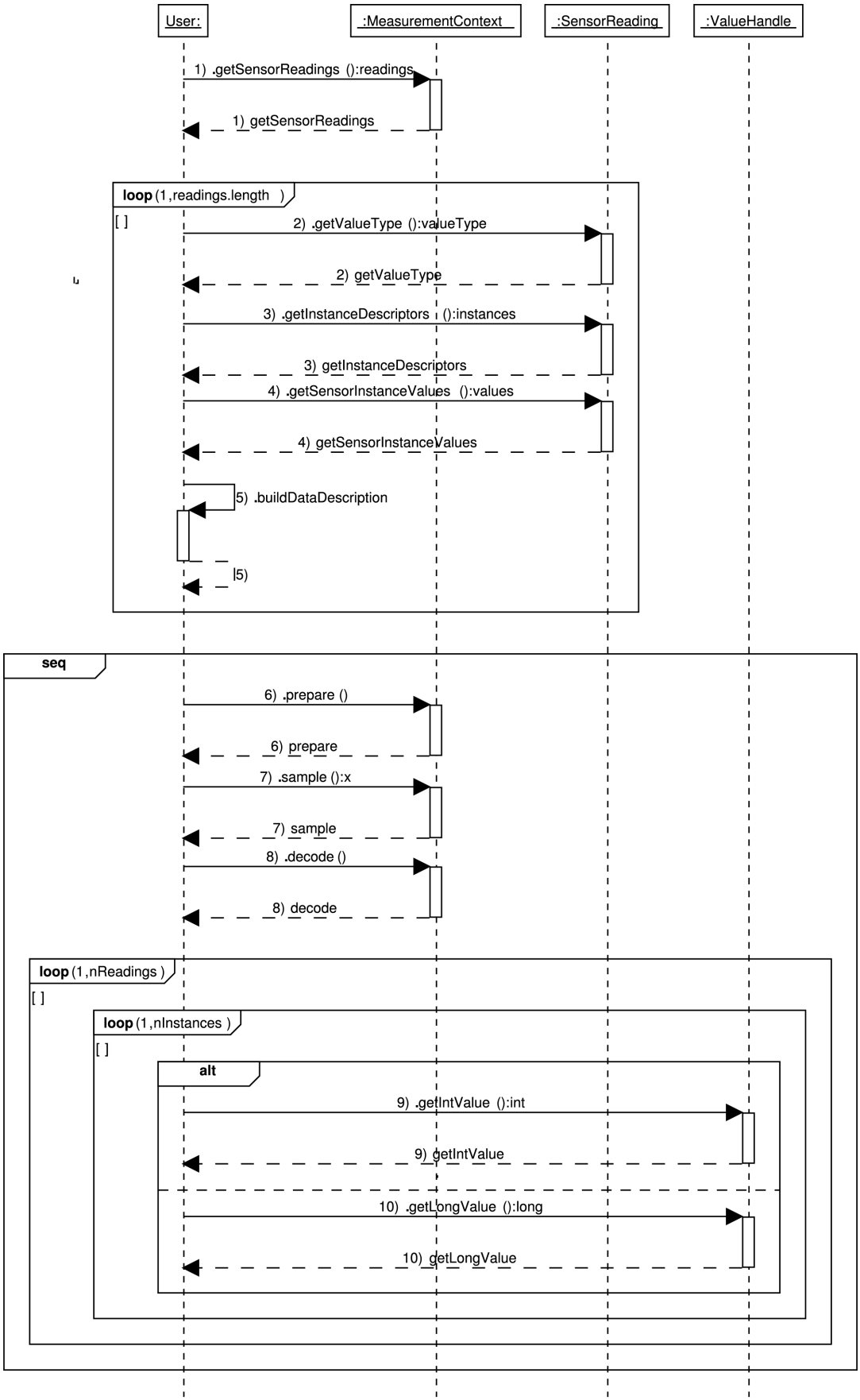


Figure 9: Accessing sensor values during measurement

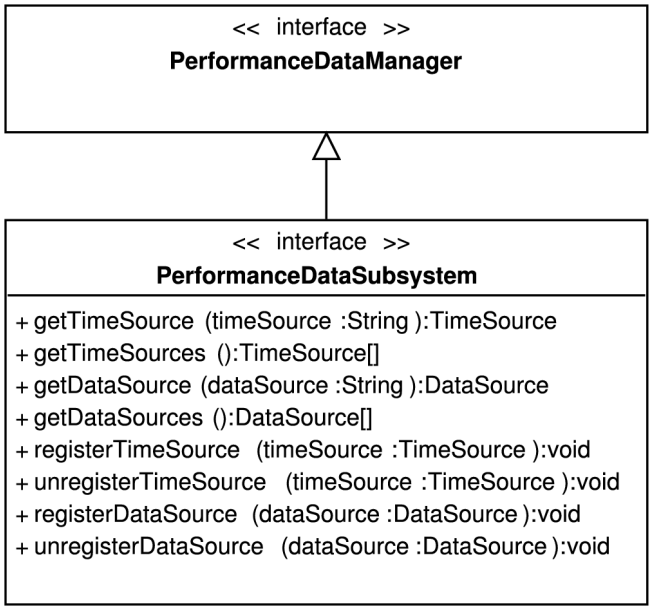


Figure 10: Internal management interface of Performance Data subsystem

4.2.2 Internal interfaces

While the external interfaces serve for querying and accessing performance data, the subsystem also has a number of internal interfaces, which are used by an implementation of a Performance Data subsystem to manage time and data source modules, obtain information on provided sensors, and coordinate access to performance data from diverse data sources. Like in the case of external interfaces, the subsystem defines management and measurement interfaces. Besides an interface to the subsystem itself, the management interfaces include also module interfaces.

Management interface

The internal management interface of the Performance Data Subsystem is depicted in Figure 10. The PerformanceDataSubsystem extends the PerformanceDataManager interface to allow keeping a single reference to the subsystem, depending on the level of functionality required by a user of the subsystem. The extended subsystem interface is rather simple, since it mainly provides functions for registering and unregistering data and time sources. The registration methods allow decoupling construction of a Performance Data subsystem instance from its configuration. A factory creating an instance may first determine which performance data sources are available on the platform and only register those that can be actually accessed. Moreover, explicit registration of data sources allows applications using the subsystem to register their own Data Source Modules providing access to application specific performance data.

Another function of the interface is to provide access to the Time Source and Data Source Modules registered in the subsystem. This is achieved by calling an appropriate getTimeSource or getDatasource method. The ability to access the Data Source modules is required to allow creating virtual Data Source Modules, which define their own set of sensors, but use other Data Source Modules to access their values, such as e.g. in case of a virtual Data Source Module providing generic sensor names portable across different platforms.

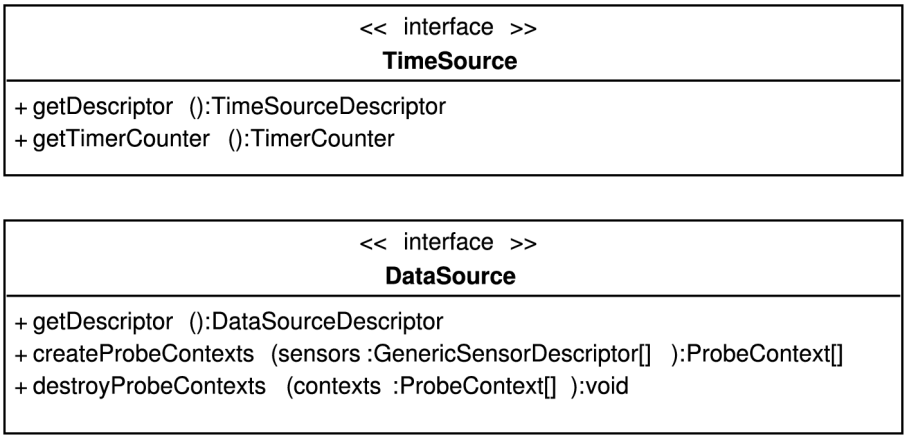


Figure 11: Module interfaces of Performance Data subsystem

Module interfaces

The management interfaces of Performance Data subsystem modules are depicted in Figure 11. Both interfaces are simple, because they only provide access to module specific information and access to measurement interfaces. Since using a Time Source Module is simple and straightforward, the TimerCounter measurement interface belongs to the external measurement interfaces presented earlier. On the other hand, the measurement interface of a Data Source Module can only access sensors from a particular data source. Therefore the module measurement interface is internally used by the subsystem to implement its external measurement interface, which allows accessing all available sensors.

The Data Source Module measurement interface can be obtained by calling the createProbeContexts method, which provides a multiple ProbeContext interfaces. The interface is conceptually similar to the MeasurementContext interface and will be described later.

Logical structure of a Data Source Module

In general, each Data Source Module is responsible for a set of sensors, provides a unified interface to the Performance Data subsystem, and there should be no reason to have specific requirements on its internal implementation. However, before describing the internal measurement interfaces subsystem, we need to introduce an additional concept related to the internal structure of a Data Source Module.

The logical structure of a Data Source Module in the context of the performance data subsystem is illustrated in Figure 12, which shows a Data Source 1 module providing the **Performance Data Subsystem** with a set of sensors and their instances. Between the module and the provided sensors, there are multiple Probe entities that are responsible for accessing the values of a particular set of sensors and their instances. The union of all sets of sensors provided by Probes represents the set of sensors provided by a Data Source Module, which is also responsible (not the Probes) for mapping sensor names to performance data provided by a particular data source.

This architecture is necessary because some data sources may only provide snapshots of performance data with the granularity of performance entities instead of individual counters or gauges. This occurs e.g. in case of the Linux *proc* and *sysfs* virtual file system,

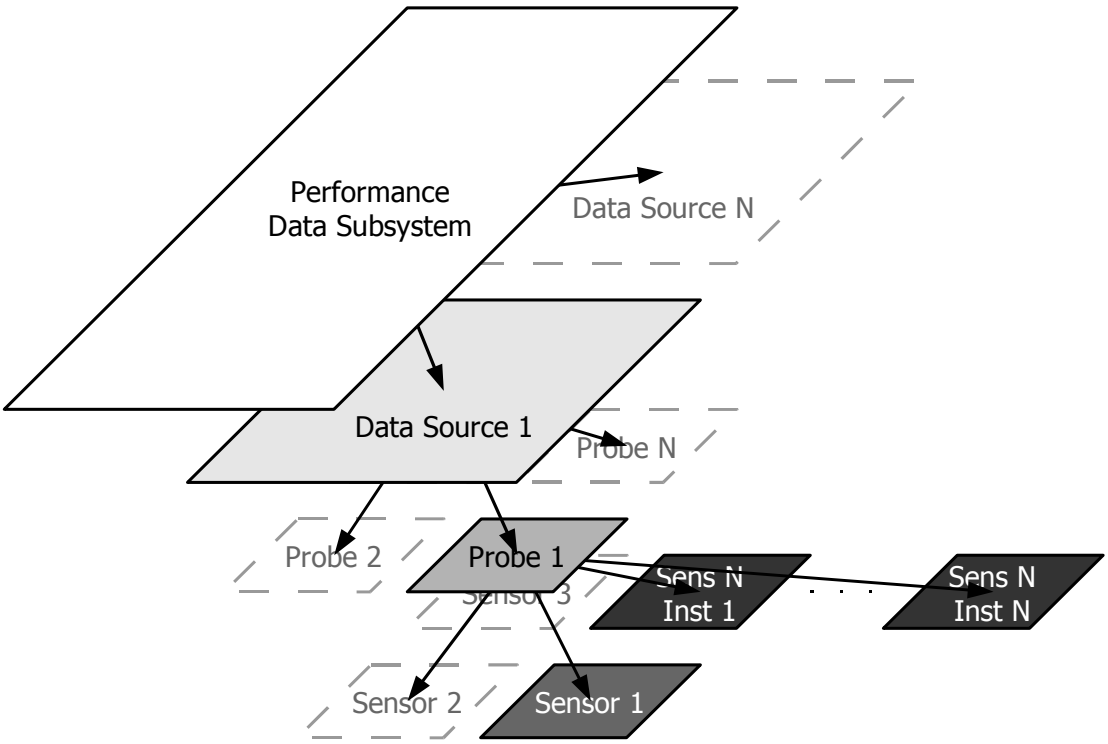


Figure 12: Structure of a Data Source Module

where certain files provide values for several counters and their instances, as well as in case of the Windows Performance Data subsystem, where performance counters and gauges are associated with a performance object and cannot be obtained individually. Moreover, each performance object can be only obtained as a whole, with all its instances, which is both time and memory consuming. To access performance data efficiently, a Data Source Module should group sensors provided by the same data source entity, take a single snapshot of its performance data and decode values of multiple sensors after performing only a single access to the data source.

The concept of Probes allows expressing the granularity of access to a performance data source within a Data Source Module. Therefore in the case of Windows Performance Data, a Data Source Module will contain be multiple Probes, each corresponding to a single performance object, while in the case of Linux virtual file systems, there will be a Probe for each file. In case of other data sources, there may be only a single Probe.

While the internal implementation of the Probe concept is the responsibility of a particular Data Source Module, it needs to expose a measurement interface for each of the entities. This is to avoid an extra level of indirection when using the module measurement interface from the subsystem implementation of the MeasurementContext interface. When creating an instance of an entity behind the interface, the subsystem first examines the sensors for which the MeasurementContext should be created and identifies all Data Source Modules that need to be contacted to obtain the sensor values. After invoking the sample (and other methods) method on the interface, it would have to invoke a similar method on each Data Source Module, which would in turn invoke a similar on its Probes, which would finally access the data source. With the measurement interfaces of individual Probes exposed, the implementation of the MeasurementContext may invoke the measurement methods directly on the probes, which in turn allows making performance data collection operations more efficient.

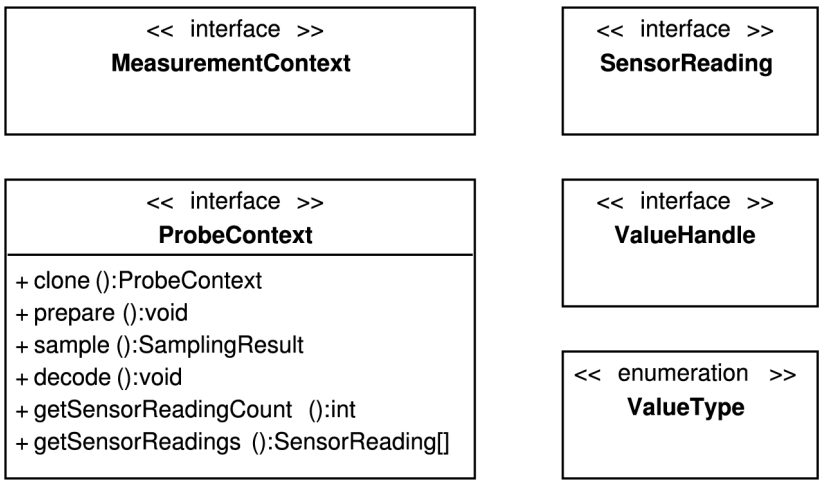


Figure 13: Internal measurement interface of Performance Data subsystem

Measurement interface

The internal measurement interface of the Performance Data subsystem is represented by the ProbeContext interface depicted in Figure 13. In principle the interface provides the same methods as the MeasurementContext interface, but represents only a part of the entire measurement context. For each Data Source Module, there may be multiple ProbeContext interfaces, corresponding to internal Probe entities responsible for gathering performance data with granularity supported by a particular data source.

Creating a MeasurementContext

Instances of ProbeContext are created by the subsystem in response to a user calling the createMeasurementContext method on the PerformanceDataManager interface, as illustrated in Figure 14. After receiving the request, the PerformanceDataManager implementation examines the sensors to be included in a measurement context to determine a set of Data Source Modules that have to be used to obtain the sensor values; this activity is denoted by the internal selectDataSources method. The PerformanceDataManager then iterates over the set of modules, and by calling the createProbeContexts method on their DataSource interface it requests ProbeContext interfaces to be created for the sensors associated with a particular module.

The next activity is internal to the implementation of a particular Data Source Module, but it is expected that the module will perform mapping of sensors to performance entities of the data source and determine a set of Probes responsible for obtaining their values. It will then use an internal method to create measurement contexts for each of the Probes, which will typically include allocating resources needed to access the particular data source entities. The collected ProbeContext interfaces will be returned to the PerformanceDataManager, which will aggregate the responses from all Data Source Modules.

Finally, the PerformanceDataManager will create a MeasurementContext instance and provide it with the collected the ProbeContext interfaces. In turn, the MeasurementContext instance will query the number of sensor readings associated with each ProbeContext, prepare its own data structures and collect all associated SensorReading instances. Then the MeasurementContext is ready and can be returned to a user. During

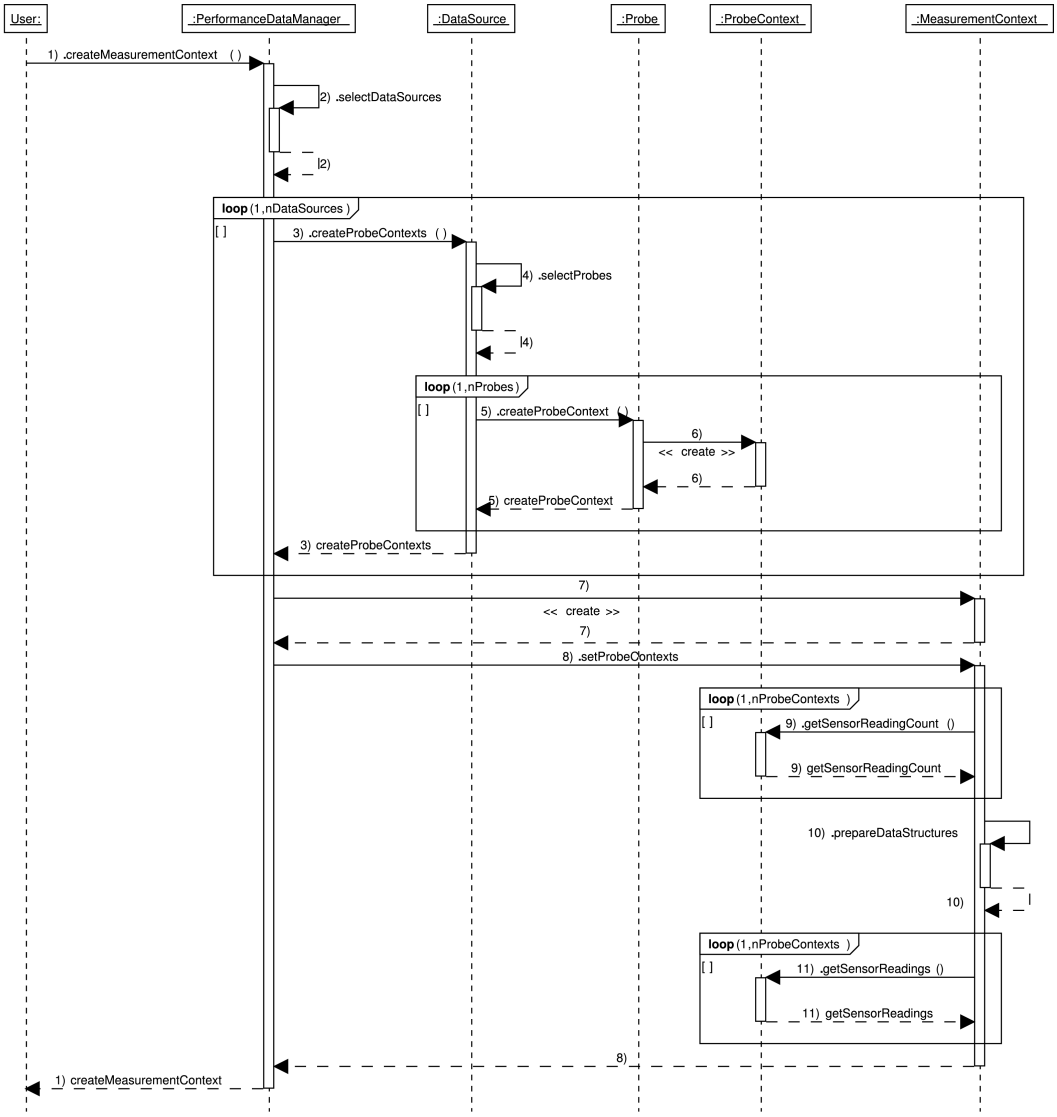


Figure 14: Creating a MeasurementContext

operation, the MeasurementContext implementation will delegate all measurement calls to all associated ProbeContext interfaces.

Creating a MeasurementContext for aliased sensors

The main purpose in exposing the subsystem internal interfaces is to allow creating Data Source Modules which define aliased sensors, such as a module providing generic, portable, sensor names of selected types of performance counters and gauges. Such aliasing module only needs to define mapping between the generic names and the names of semantically equivalent sensors provided by other Data Source Modules. When created, the module will query the Performance Data subsystem and determine which generic sensors can be provided given the currently available sensors. After registering with the Performance Data subsystem, the generic sensors will be available to a user.

When a user requests a MeasurementContext using the generic sensor names, the PerformanceDataManager will contact the aliasing module with a list of sensors and require a set of ProbeContext interfaces in return, as depicted in Figure 14. Since the

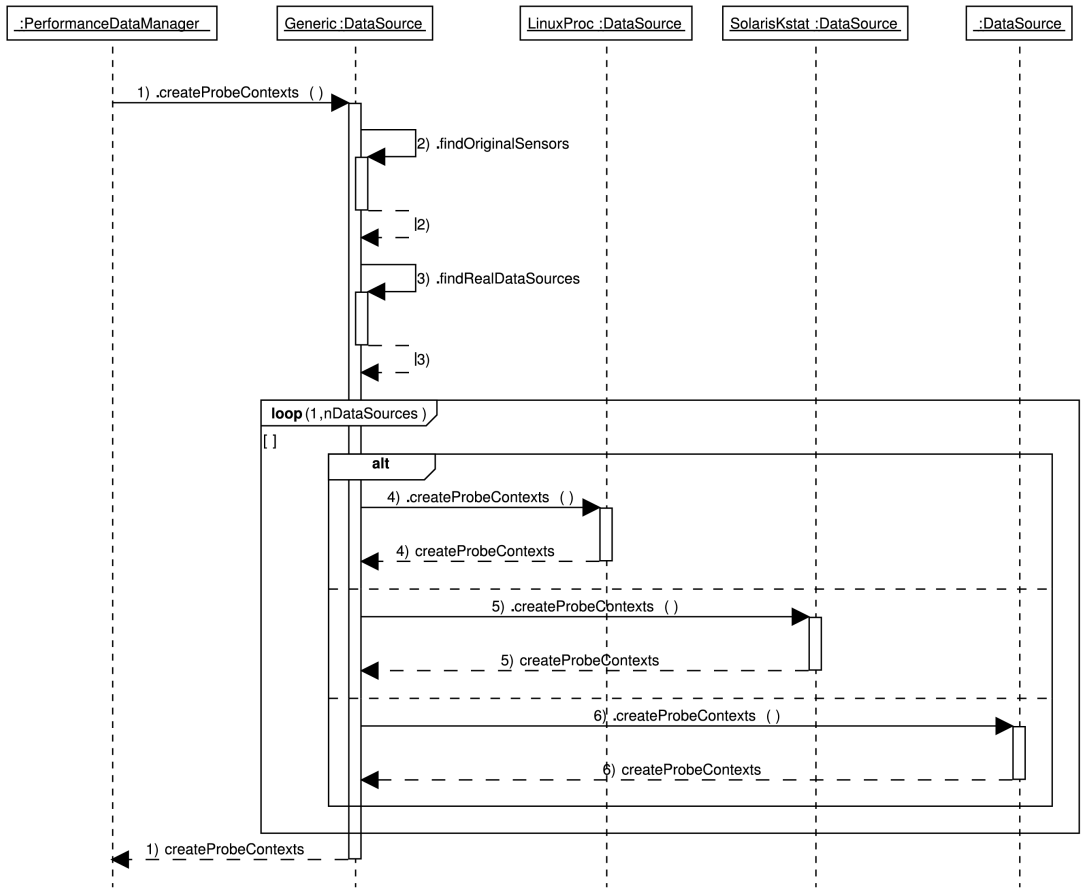


Figure 15: Creating MeasurementContext for aliased sensors

aliasing module has access to other Data Source Modules, it may request forward the request to other modules and provide their ProbeContext interfaces instead, as illustrated in Figure 15.

After receiving the createProbeContexts call, the aliasing module will consult its mapping structures to find the names of real sensors corresponding to the generic sensors. It then locates the Data Source modules responsible for the real sensors and calls the createProbeContexts method on their DataSource interfaces to obtain ProbeContext instances which will provide the values of requested sensors. After aggregating the responses from different modules, the aliasing module returns the resulting set of ProbeContext interfaces to the PerformanceDataManager, which uses it to construct a MeasurementContext instance.

Subsequent measurement operations invoked on the MeasurementContext interface will be delegated directly to the Probes from individual Data Source Modules responsible for a particular set of real sensors, completely avoiding the aliasing module. However, this approach is only possible when all semantically equivalent sensors provide values with identical data types. If this condition is not satisfied, the aliasing module may choose to provide type conversion for a set of sensors, in which case it would return its own ProbeContext interface and delegate the measurement calls to the respective data sources itself. It would also provide its own SensorReading interface and ValueHandle instances performing type conversion for selected sensors.

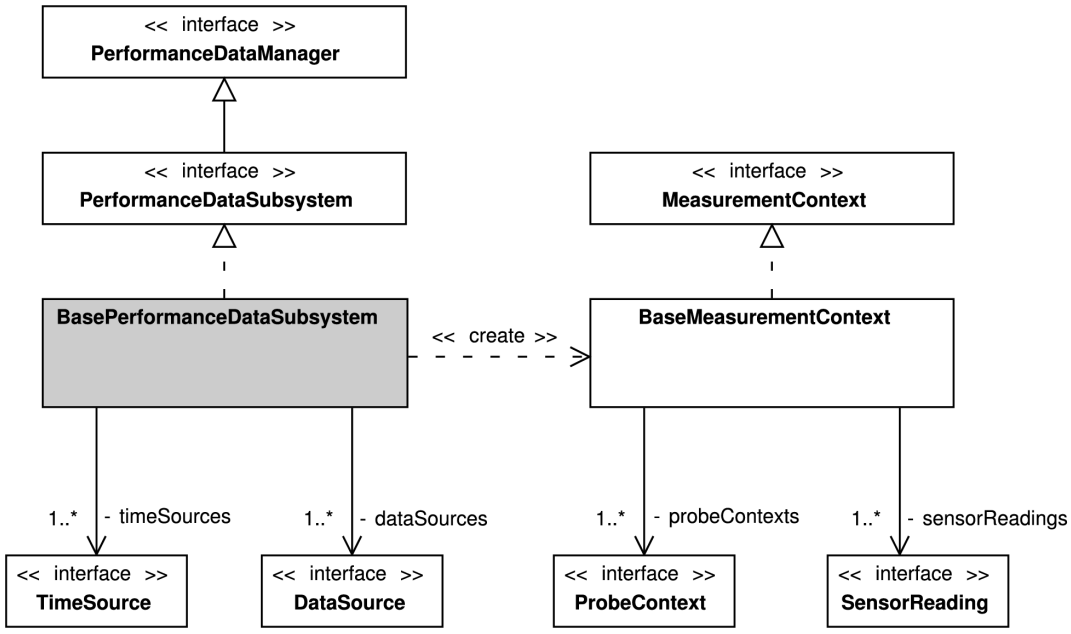


Figure 16: Architecture of Subsystem Public Interface

4.2.3 Subsystem architecture

With the basic concepts and interfaces in place, we can now provide a final overview of the Performance Data subsystem architecture. For actual implementation, the following models need to be further elaborated and serve mainly as structural guide.

Subsystem Public Interface

The architecture of the Subsystem Public Interface part of the Performance Data subsystem is depicted in Figure 16. Since the real functionality of this part is to aggregate information and provide access to different data and time sources, its architecture is very simple. A single class, such as BasePerformanceDataSubsystem, may implement the browsing and management capabilities, while another class, such as BaseMeasurementContext, may implement the delegation functions required for measurement operations.

Time Source Module

The architecture of a Time Source Module depicted in Figure 17 is trivial and has been included mainly for completeness sake. The module will be typically implemented by a single class, which will be responsible for providing the management, information, and measurement interfaces. For each time source, there may be different class, as depicted in the figure.

Data Source Module

The architecture of a Data Source Module depicted in Figure 18 is the most complex of the three. Even though the subsystem does not have to care about the internals of a particular Data Source Module because it only accesses it through a set of interfaces, the figure shows the envisioned structure of typical Data Source Module, along with responsibilities

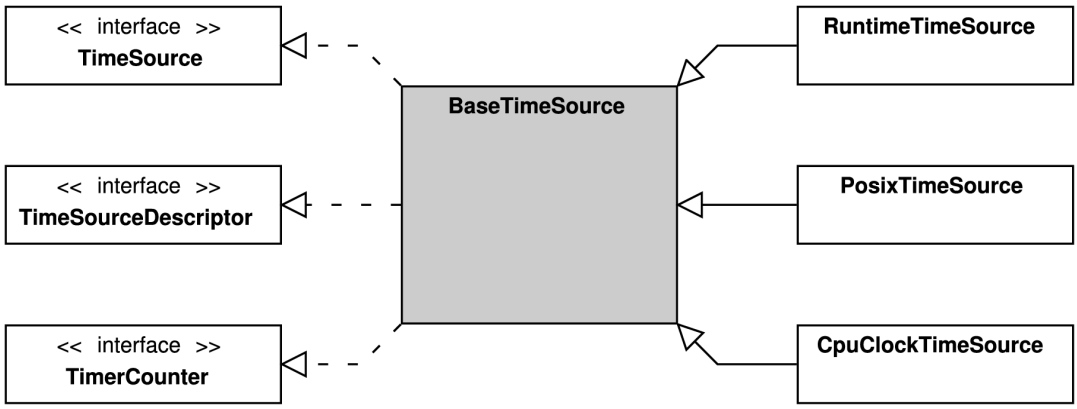


Figure 17: Architecture of Time Source Module

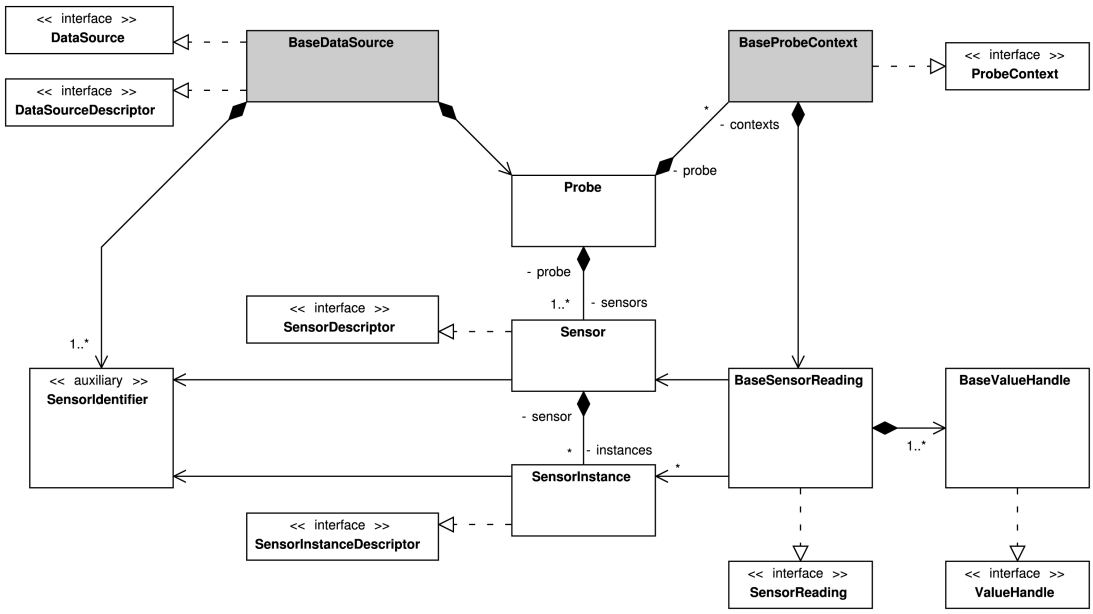


Figure 18: Architecture of Data Source Module

for implementing particular interfaces. With the exception of the Probe class, all entities are connected to the subsystem through one of its defined interfaces. The left side of the figure contains the management part of a module, while the right part contains the measurement part. The entities in the middle serve are shared by both parts.

4.3 Measurement Infrastructure Subsystems

The Performance Data subsystem described in previous section represents the key component of the measurement infrastructure. While other parts of the infrastructure are important as well, the Performance Data subsystem provides support for the basic concept of collecting performance data associated with performance events. The measurement infrastructure will mainly use the Subsystem Public Interface part of the Performance Data subsystem, represented by the PerformanceDataManager, MeasurementContext, and other external interfaces, but it may decide to implement computation of aggregate statistics and other derived metrics by registering a special Data Source Module in the subsystem.

General architecture of the measurement infrastructure, along with the responsibilities of individual subsystems and interactions among them, has been already described in Section 4.1 (see Figure 1 for reference). This section provides more detail on the design of the measurement infrastructure subsystems, which have been split into measurement and management groups depending on the source driving their activity.

4.3.1 Measurement: application driven subsystems

The first group represents subsystems that are driven by application execution, more precisely the performance events related to the execution. The events are typically emitted by a standalone benchmark application, by performance instrumentation code, or by an application itself. The important point is that the subsystems execute in the application context and become part of application execution, because an application cannot proceed until the performance events are processed.

Together, the subsystems in this group resemble a processing pipeline, even though some of them can be decoupled from the application context and process performance events asynchronously. The Event Sources and Event Processing subsystems are responsible for the synchronous part of execution, while the Data Storage subsystem can operate asynchronously.

Event Sources

Event sources correspond to runtime entities responsible for emitting performance events. In general, these entities will be external to the measurement infrastructure and will be typically located in application performance instrumentation or in a benchmark application.

To use an external performance event source with the measurement infrastructure, the entity responsible for emitting performance events has to first register with the infrastructure. During registration, it will provide the infrastructure with a management interface that will be used by the infrastructure to query information about supported performance events and to control the event source. Since the event source management interface may be too complex for casual use in simple benchmark applications, the infrastructure provides a few internal event sources with a simple interface, which can be easily obtained through the infrastructure management interface.

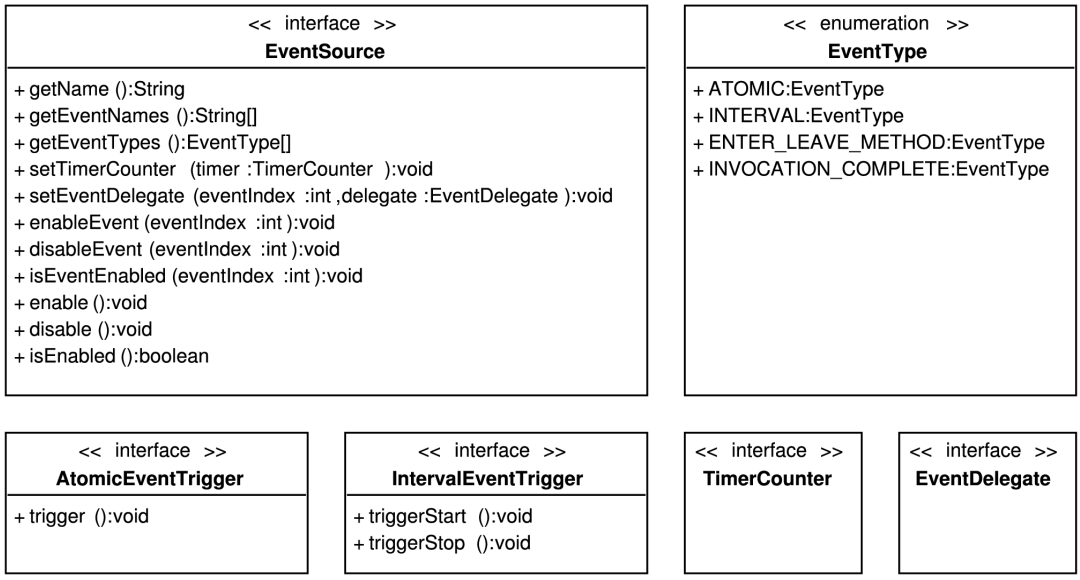


Figure 19: Event Sources interfaces for external entities

Interfaces for external entities

The external event source management interface as well as the interfaces to the simple internal event sources is shown in Figure 19. The EventSource interface is primarily intended to be implemented by external performance instrumentation code. As such it has been intentionally designed to use only simple types to avoid creating many instances of small-sized informational objects, which would only increase the memory overhead of the instrumentation.

The EventSource interface contains three sets of methods. The first set is represented by the getName, getEventNames, and getEventTypes methods, which provide information about an event source and the events it supports. Each event is associated with a name, which is typically derived from its location in an application, and EventType, which defines how a particular event should be processed by the infrastructure.

The second set of methods, represented by the setTimerCounter and setEventDelegate methods, serves to provide an event source with interface references it requires from the infrastructure. An infrastructure may provide an event source with a common time source in form of a TimerCounter interface, which allows an event source to obtain timing information compatible with other event sources and the rest of the infrastructure. However, more important is to provide an event source with an EventDelegate interface reference for each event it supports. These will be used to submit event notifications to the infrastructure.

And finally the third set of methods consists of the enable, disable, enableEvent, disable-Event, and similar methods, which provide the infrastructure with control over the events emitted by an event source. The enable and disable methods serve as a master switch and allow the infrastructure instruct an event source whether it should or should not emit any performance events. The enableEvent and disableEvent methods serve for fine-grained configuration and allow the infrastructure to instruct an event source to only emit particular events when enabled.

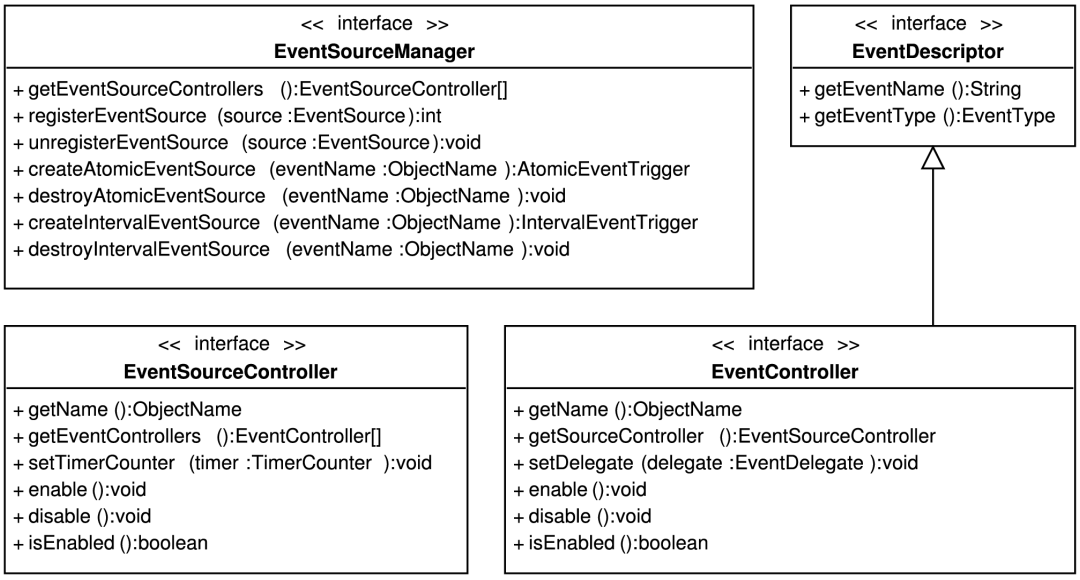


Figure 20: Event Sources interfaces for Infrastructure Management

The argument of the event-specific methods is an event index which corresponds to the order of event names returned by the getEventNames method. The index represents a local identifier specific to each event source and is intended to let the event sources manage their configuration in simple way.

The EventSource interface is intended to be used by performance instrumentation code and allows using a single event source entity for a group of related events, such as e.g. events corresponding to interface method invocations. However, because of the support for event groups and event source configuration, the interface is relatively complex for casual use, when only a few unrelated performance events need to be defined. To simplify using the infrastructure in such cases, the Event Sources subsystem provides an internal implementation of two simple event sources, which can be used to emit single-event notifications through the AtomicEventTrigger and IntervalEventTrigger interfaces.

Interfaces for Infrastructure Management subsystem

Since the EventSource interface intended for external event sources is rather low-level, it is not very suitable for use by the Infrastructure Management subsystem. The Event Sources subsystem therefore provides high-level interfaces for event sources and their performance events. In addition, it provides a management interface to the Infrastructure Management subsystem. These interfaces are shown in Figure 20.

The high-level facade for event sources and associated events is provided by the EventSourceController and the EventController interfaces. Each external event source is represented by a single EventSourceController, which provides access to multiple instances of the EventController interface representing the events supported an event source. Each controller interface contains only methods corresponding to the entity it represents. In particular, the EventController interface allows enabling or disabling individual events without having to use event indexes. The EventDescriptor interface provides only information-level access to a particular event. This interface can be passed outside the infrastructure to provide users with information about available events.

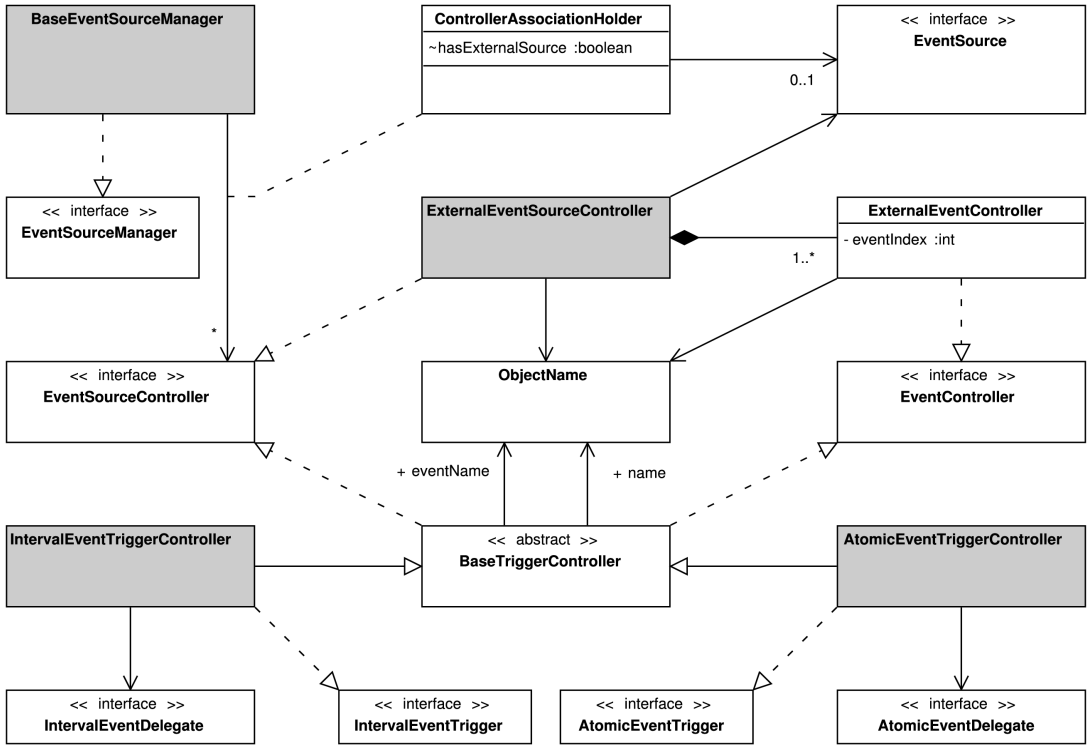


Figure 21: Event Sources subsystem architecture

Finally the EventSourceManager interface provided to the Infrastructure Management subsystem allows querying the available event sources, registering new event sources, and creating instances of the simple event triggers for use in simple applications. With the exception of the getEventControllers method, the methods of the EventSourceManager interface are expected to be used by users of the infrastructure. The Infrastructure Management subsystem is therefore responsible for exposing a suitable interface to the functionality they provide.

Internal architecture and operation

The internal structure of the Event Sources subsystem is depicted in Figure 21. The diagram captures only the essential concepts and responsibilities for implementing the various interfaces. The upper half of the picture is related to event source management. The BaseEventSourceManager class implements the EventSourceManager interface and is responsible for event source registration and for management of other classes.

The ExternalEventSourceController and ExternalEventController classes implement the high-level facade for external event sources, using the EventSource interface provided during registration. The IntervalEventTriggerController and AtomicEventTriggerController classes implement the simple event sources and also implement the controller interfaces to maintain the appearance of an event source to the Infrastructure Management. Since they are intended for specific event types, each requires a corresponding type of event delegates.

When the BaseEventSourceManager class receives a request to register an event source, it creates an instance of ExternalEventSource controller, which in turn creates instances of ExternalEventController classes for each event supported by an event source. The manager class then creates a new instance of the ControllerAssociationHolder class to

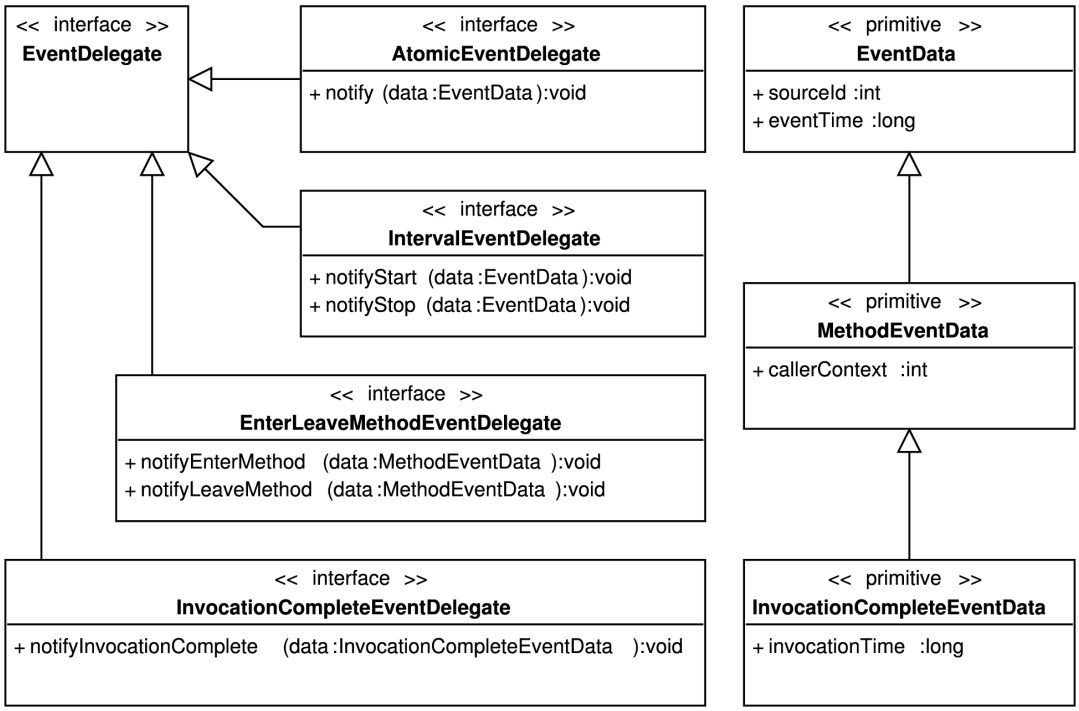


Figure 22: Event Processing interfaces for Event Sources

maintain the association between the event source interface being registered and the newly created event source controller. When a simple event trigger is requested, the manager class instantiates the appropriate event trigger class and registers it within its structures. Since the event trigger classes do not implement the external EventSource interface and instead directly implement the controller interfaces, there is no need to create additional controller classes.

Event Processing

The actual collection of performance data associated with events related to application execution is realized by the Event Processing subsystem. In contrast to event sources, this subsystem is completely internal to the infrastructure and provides external event sources with entry points to report performance events to the infrastructure.

The entry points are represented by event delegates, which process a particular type of events, and which are assigned to each event supported by an event source.

Interfaces for Event Sources subsystem

For each event type, there has to be a specific event delegate, which provides an interface for different event notifications. The interfaces of event delegates for the currently envisioned event types are shown in Figure 22.

The AtomicEventDelegate and the IntervalEventDelegate delegates are intended for the simple event triggers provided to users by the measurement infrastructure itself. These delegates only expect basic type of EventData, which consists of event source identifier and a time stamp related to event occurrence. The other two delegates, the EnterLeave-MethodEventDelegate and the InvocationCompleteEventDelegate, are intended for use with event sources emitting events related to method invocations, typically found in

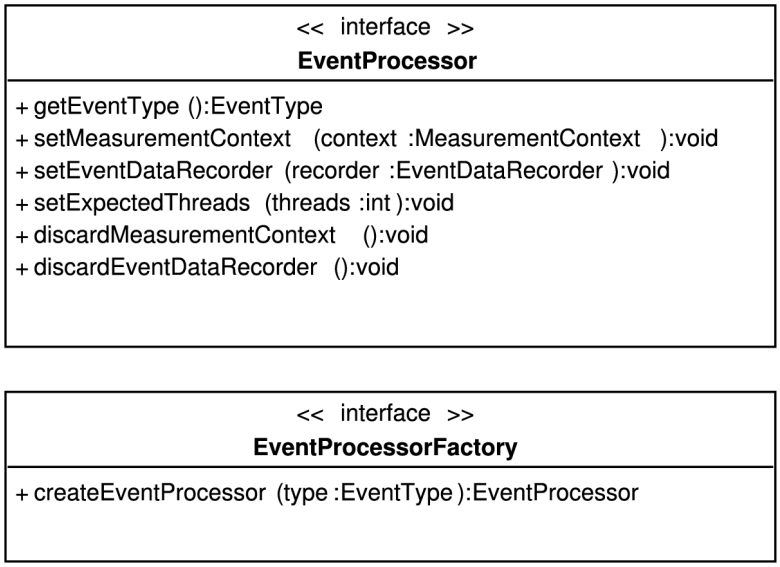


Figure 23: Event Processing interfaces for Infrastructure Management

performance instrumentation. In addition to basic event data, the delegates intended for method level performance instrumentation contain additional information such as caller context, or the time of method invocation start in case of the InvocationCompleteEvent-Delegate. While they both provide similar kind of data, the EnterLeaveMethodEvent delegate can react differently to start of method invocation and its return, even though this may mainly concern the Data Storage subsystem.

Interfaces for Infrastructure Management subsystem

In addition to the event delegate interfaces provided to the Event Sources subsystem, the Event Processing subsystem provides management interfaces shown in Figure 23 to the Infrastructure Management subsystem. The Event Processing entities implementing the event delegate interfaces assume a native role of event processors when interfacing with Infrastructure Management.

The factory interface serves creating event processors for specific event types, and each event processor entity implements the EventProcessor interface, which is used by the infrastructure to configure an event processor before assigning it (in form of event delegate) to a particular event. Configuring an event processor entails providing it with a reference to a base MeasurementContext (see Figure 7 in Section 4.2.1), which maintains the association between an event and performance data that should be collected when an event occurs, the expected number of threads that will be emitting performance events through the event delegate interface, and a reference to a Data Storage entity which will be responsible for processing and storing the data associated with each event.

After configuration, an event processor may adjust its internal data structures to accommodate multiple execution contexts, such as creating a pool of Measurement-Context clones to avoid synchronization on a single instance. It will also register all its MeasurementContext instances (including the initial one) with the assigned Data Storage entity to let it examine the structure of performance data (in form of sensors), associated with a MeasurementContext, prior to operation. Upon receiving an event notification on its event delegate interface, an event processor will use an available MeasurementContext to sample values of sensors associated with an event. Unless configured otherwise, an

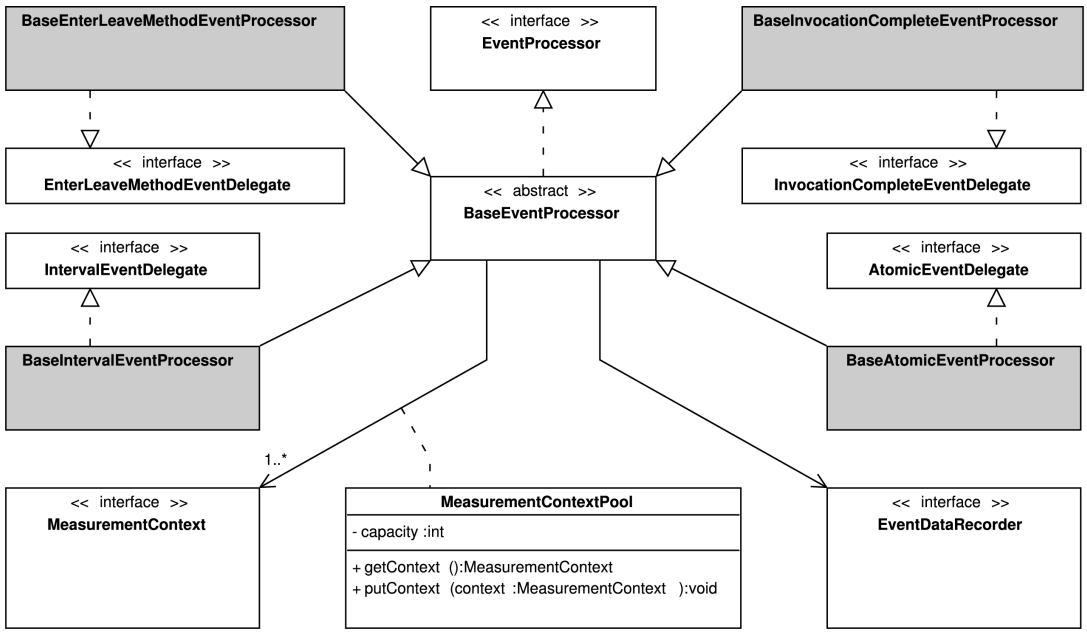


Figure 24: Event Processing subsystem architecture

event processor may only use (and synchronize on) a single MeasurementContext. After taking a snapshot of sensor values, it will submit the MeasurementContext for processing and storage.

At this point, it is possible to return to the application and let the Data Storage subsystem process the data asynchronously. However, this would require adding a bidirectional queue between the two subsystems for holding MeasurementContext references, both available for use and submitted for storage. Since this is mainly a quality of implementation issue, we will not elaborate it further.

Internal architecture and operation

The internal structure of the Event Processing subsystem is depicted in Figure 24. Compared to the Event Sources subsystem, it is simpler and consists mainly of multiple classes representing event processors for different event types. The abstract base class captures the key associations common to all event processor classes. The event processor factory class and related associations have not been included in the figure for clarity.

All event processor classes implement the EventProcessor interface through which they interface with the Infrastructure Management subsystem. Each event processor will also contain a set of references to MeasurementContext instances and a reference to an EventDataRecorder instance responsible for recording events and associated performance data. The MeasurementContextPool association class captures the concept of creating multiple MeasurementContext clones for multiple execution contexts, which is necessary to avoid synchronizing on a single MeasurementContext instance in multi-threaded environment.

Initially only the base MeasurementContext will be registered with an EventDataRecorder. After receiving a hint from the infrastructure concerning the number of threads expected to be generating performance events, an event processor will create additional MeasurementContext clones and register them with the event data recorder.

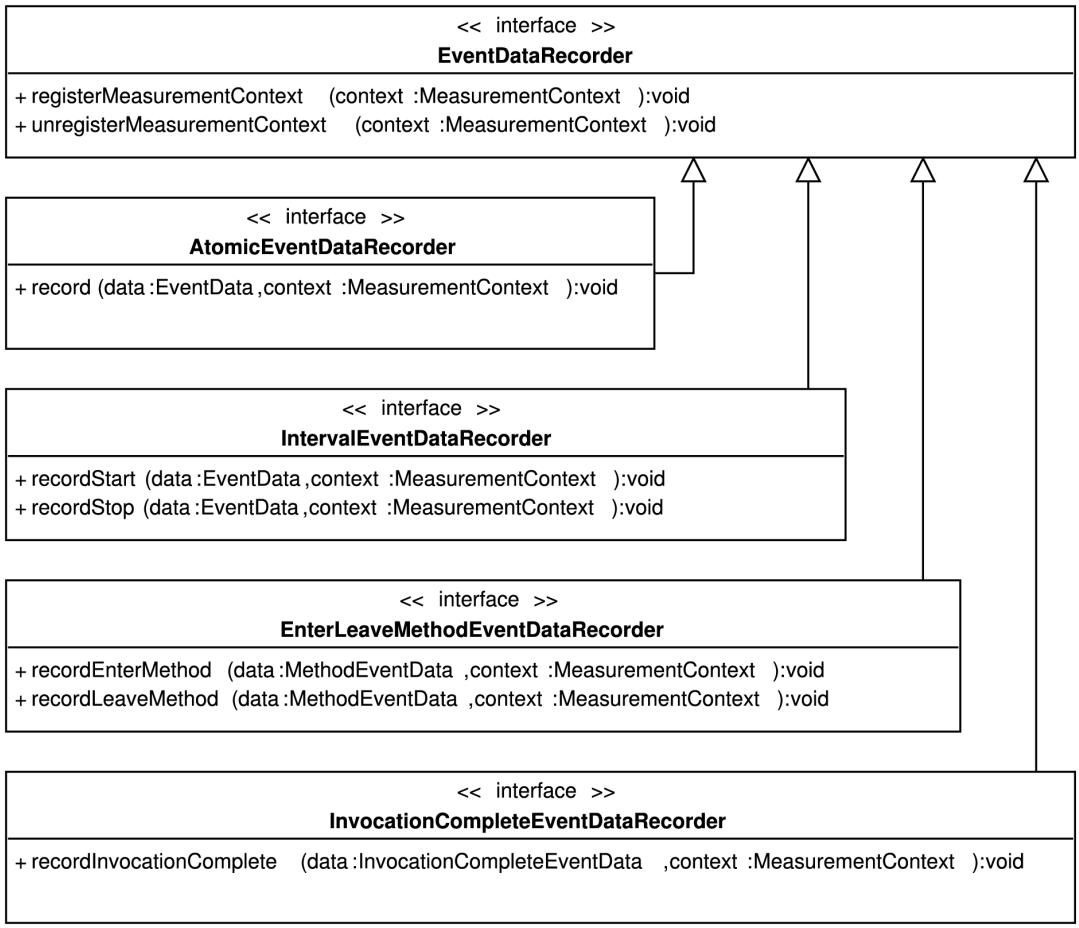


Figure 25: Data Storage interfaces for Event Processing

Data Storage

After taking a snapshot of performance data associated with a performance event, an event processor will submit the event data provided by an event source, along with the sensor values associated with a MeasurementContext to the Data Storage subsystem. The data are passed to event data recorders, which store event records along with performance data and update aggregate values based either on sensor values associated with a MeasurementContext, or values contained in event data associated with a particular event type.

Interfaces for Event Processing subsystem

The event data recorder interface provided by Data Storage subsystem to the Event Processing subsystem is similar to that of event delegates. For each event type, there is a different event data recorder, which provides interface for different event notifications, as shown in Figure 25.

As in case of the event delegates, the AtomicEventDataRecorder and the IntervalEvent-DataRecorder recorders are intended for event processors associated with simple event triggers, while the EnterLeaveMethodEventDataRecorder and the InvocationComplete-EventDataRecorder recorders are intended for event processors associated with complex, multi-event data sources providing events related to method invocations.

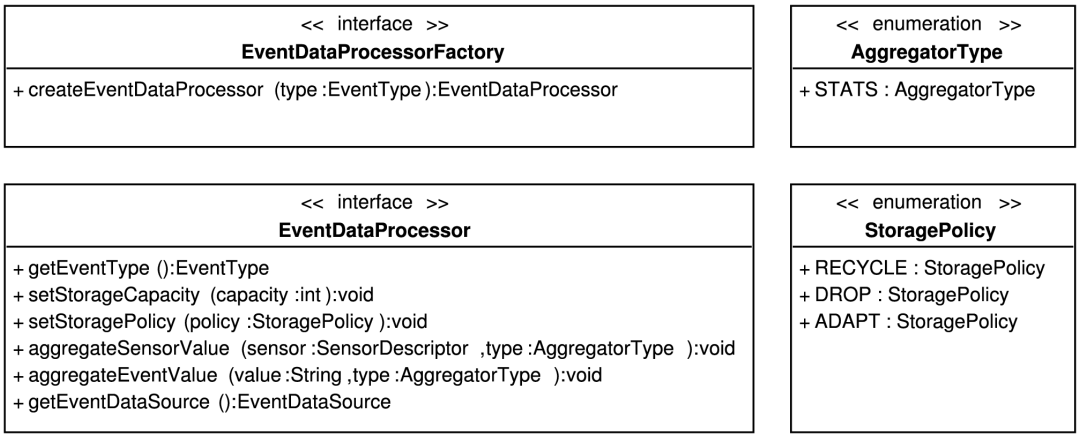


Figure 26: Data Storage interfaces for Infrastructure Management

As shown in Figures 22 and 25, each event delegate and event data recorder is associated with a different type of event data, with the data types corresponding to event types. The values contained within the EventData classes can be used by event data recorders to derive additional data values with an event, but what values can be derived depends on a particular event type. An AtomicEventDataRecorder only receives event time stamp in event data and since atomic events are unrelated to each other, it can at most count the events it has processed or determine the rate at which events are occurring. The InvocationCompleteEventDataRecorder is in similar situation in that it only supports a single event notification, but the event data it is provided with contain also a time stamp of method invocation. This allows, along with the time stamp of the event itself, to derive the method invocation duration from the event data. The remaining two event data recorders, the IntervalEventDataRecorder and the EnterLeaveMethodEventDataRecorder, are similar in that they both support event notifications for two related events. This allows a data recorder to keep an internal state and relate event data from both notifications, and derive e.g. a time interval between the two events, which in the case of the method event recorder has semantics of invocation duration.

In contrast to the EventDelegate interface, the EventDataRecorder interface is not empty and provides method that allows an event processing entity to register the MeasurementContext instances it uses for collecting performance data associated with an event. This registration serves to allow an implementation of a data recorder to examine the structure of performance data with each MeasurementContext and prepare its own data structures for efficient access to the data.

Interfaces for Infrastructure Management subsystem

Along with the event data recorder interfaces provided to the Event Processing subsystem, the Data Storage subsystem provides another set of interfaces shown in Figure 26 to the Infrastructure Management subsystem. Similar to the Event Processing subsystem entities, the Data Storage subsystem entities implementing the event data recorder interfaces assume a native role of event data processors when interfacing with the Infrastructure Management subsystem.

The factory interface allows creating event data processors for specific event types, each implementing the EventDataProcessor interface. The interface used by the Infrastructure Management subsystem to configure an event data processor before assigning it (in form of event data recorder) to a particular event processor responsible for processing events

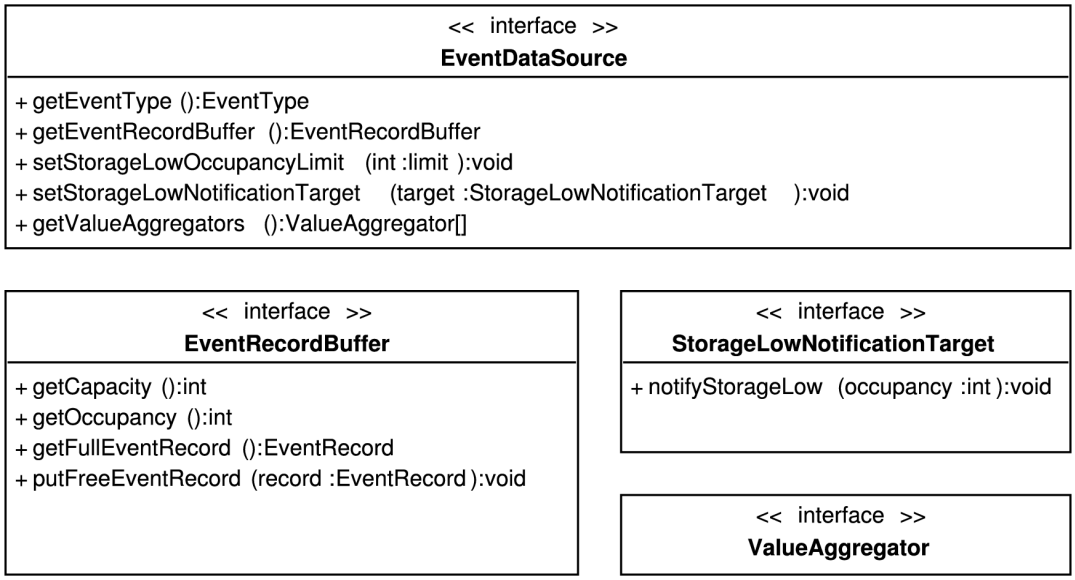


Figure 27: Data Storage interfaces for Data Delivery

of the same type. The configuration of an event data processor entails setting the number of event records that should be stored along with a storage policy which defines the behavior of data event processor in case the storage space is exhausted. The envisioned storage policies include recycling event records, thus discarding older performance data, ignoring new event data provided by event processors, or adapting the storage to accommodate all data.

In addition to event records containing event and performance data, an event processor is expected provide support for collecting aggregate data based on the performance data associated with individual events. The event processor interface provides the aggregateSensorValue and aggregateEventValue methods that allow the Infrastructure Management to configure an event processor to aggregate sensor or event specific values.

Interfaces for Data Delivery subsystem

In contrast to the Event Processing subsystem, the Data Storage subsystem also provides a set of interfaces shown in Figure 27 to the Data Delivery subsystem. The EventDataSource interface can be obtained by calling the getEventDataSource interface method on the EventDataProcessor interface and allows the Infrastructure Management to provide Data Storage subsystem entities with a limited access to event data processors. The EventDataSource interface only provides access to the data stored within an event data processor.

The getEventRecordBuffer method returns an interface to a buffer used by event data processor to store event records. The EventRecordBuffer basically provides the consumer part of buffer access interface, which is expected to be used by a Data Delivery entity to obtain and deliver performance data to a consumer. The pair of setStorageLowNotificationTarget and setStorageLowOccupancyLimit methods allows a Data Storage entity configure an event data processor to notify the entity when a certain number of records has been collected. This allows the Data Storage entity to avoid excessive polling on an EventRecordBuffer.

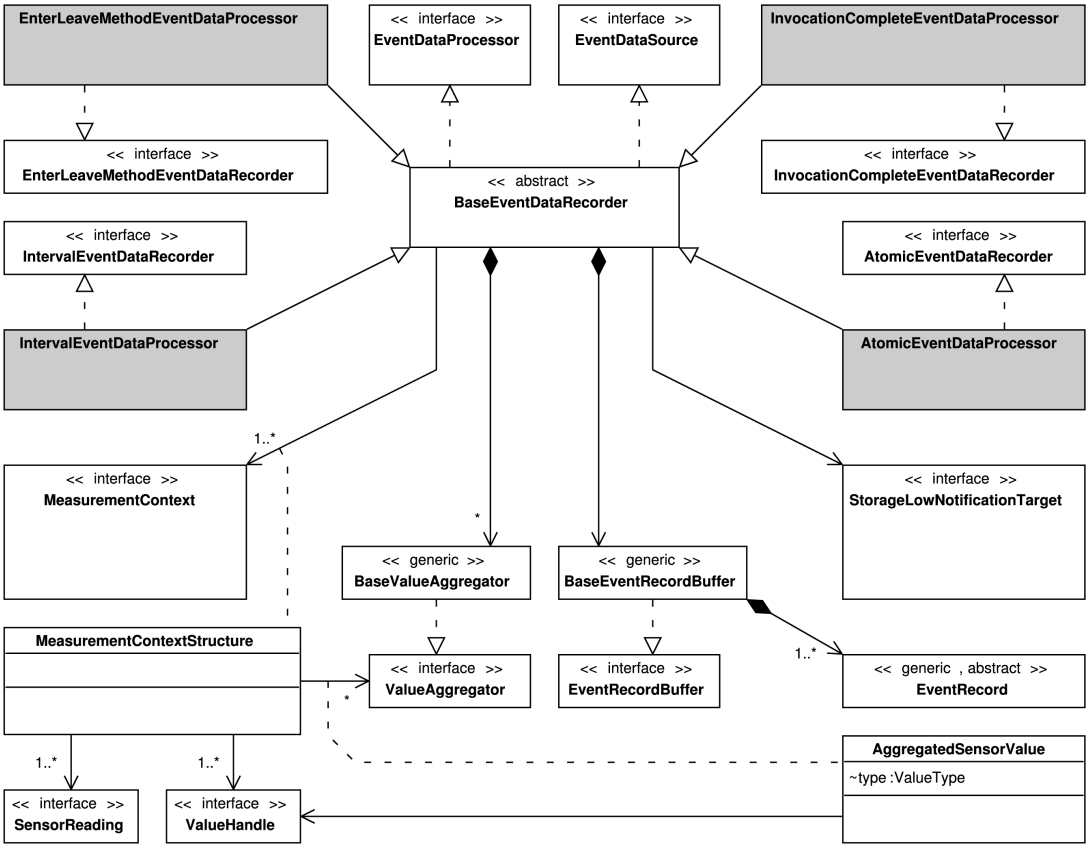


Figure 28: Data Storage subsystem architecture

The getValueAggregators method returns multiple ValueAggregator interfaces, which can be used to access aggregate data based on specific event or sensor value. However, value aggregators are in general entities computing potentially multiple values from a series of provided values. Conceptually, they correspond to performance entities providing multiple sensors.

Since they are intended to be created dynamically, this would be very similar to designing a generic Data Source Module which would allow creating user defined sensor groups and sensors with values based on provided data. Therefore to properly integrate value aggregators into the infrastructure, there would need to be additional subsystem providing different types of aggregators, with its own management and data access interfaces. If done properly, the subsystem should allow exposing the values provided by aggregators through the Performance Data subsystem by implementing the subsystem DataSource interface. However, this would only increase the complexity of the infrastructure, and since value aggregators are not essential (while collecting event records is), we leave this design work for future. Consequently, we do not provide additional details on the ValueAggregator interface.

Internal architecture and operation

The internal architecture of the Data Storage subsystem shown in Figure 28 is similar to that of Event Processing subsystem, with additional entities to accommodate the actual storage and processing of event data. The upper half of the diagram is structurally similar to the architecture of the Event Processing subsystem, because for each event type, there have to be both an event processor and an event data processor. Consequently, there are

multiple classes implementing the various event data recorder interfaces, and a base class capturing the key associations common to all event data processor classes. The event data processor factory class and related associations have not been included in the figure for clarity.

In addition to specific event data recorder interfaces, all the event data processor classes implement the EventDataProcessor and EventDataSource interfaces provided to other subsystems. Each will also contain potentially multiple value aggregators and a single event record buffer, which is a generic data structure intended for holding performance data as well as data specific to different event types. To speed up access to performance data, each event data processor will keep track of multiple MeasurementContext instances, registered by an event processor through the EventDataRecorder interface methods.

The MeasurementContextStructure association class represents the concept of exploring each registered MeasurementContext prior to operation, which allows preparing internal structures for traversing sensor values and copying them to an event record. The association class aggregates all SensorReading and ValueHandle instances associated with a particular MeasurementContext instance. When requested to store an event record with associated performance data contained in a MeasurementContext instance, an event data processor will lookup the MeasurementContext instance and use the aggregated ValueHandle instances to access the data without having to iterate over SensorReading instances. The data can be also obtained by careful swapping of SensorReading storage buffers, but that is a (non-trivial) quality of implementation issue that would need to be addressed only for very high-volume collection of performance data.

The AggregatedSensorValue association class is responsible for associating ValueHandle instances corresponding to particular sensors or sensor instances with an aggregator responsible for computing aggregate data based on a series of sensor values. This association has to be maintained for each registered MeasurementContext instance. This allows a performance data processor to easily provide aggregators with new data, without having to locate the ValueHandle instances corresponding to values of particular sensors.

There is a strong coupling between the Event Processing and Data Storage subsystems, which is evident from the need to define similar entities corresponding to a particular event type, and from the need to register MeasurementContext instances used by event processing entities with the corresponding event data processing entities. This suggests that the two subsystems should encapsulated in a single subsystem, which would coordinate the management of MeasurementContext instances, and manage the life cycle and association between the two types of entities. Consequently, the Infrastructure Management would be provided with only a single life cycle management interface and would be only responsible for associating delegates with event sources and event data sources with Data Delivery entities. However, even though the current architecture exposes more details than necessary, we do not consider it to be a major design flaw.

4.3.2 Management: user driven subsystems

The other group of measurement infrastructure subsystems represents those driven by users of the infrastructure. The user-initiated activities include creating or registering event sources, browsing available performance data and events, associating performance events with performance data, enabling and disabling particular events, configuring the storage capacity of event records, and selecting and configuring a suitable data delivery method.

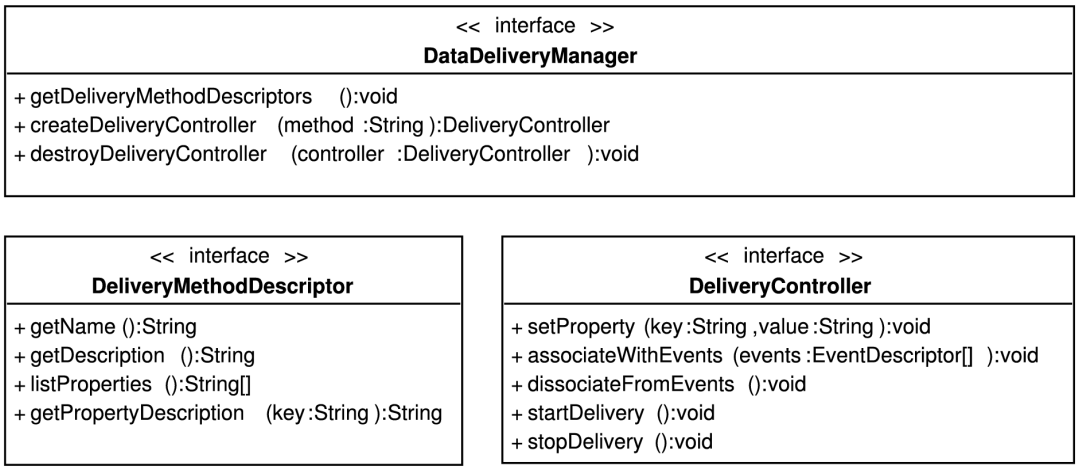


Figure 29: Data Delivery interfaces for Infrastructure Management

However, there are different kinds of users, each kind requiring different functionality. An application performance instrumentation code represents an infrastructure user mainly interested for registering event sources and reporting performance events. A simple benchmark application requires similar functionality, but typically it will also create association between performance events and performance data. A configuration console is yet another infrastructure user, interested in obtaining information about all internal entities and providing a standalone configuration interface.

In part, the required functionality is provided by the management parts of the application driven subsystems and only needs to be exposed in an appropriate manner, while the remaining functionality has to be provided by the user-driven subsystems. Their activities are typically executing in the context of infrastructure users, which is the case of the Infrastructure Management subsystem, but in some cases a subsystem main execute in a separate context, which is the case of the Data Delivery subsystem.

Data Delivery

The Data Delivery subsystem is responsible for transporting event and performance data from the memory store located in the Data Storage subsystem. This is useful for long running application or measurement experiments expected to amount significant amounts of performance data. Allocating memory for all the data is typically infeasible, especially in long running production application, therefore the Data Storage memory-based buffers need to be periodically emptied to free up space for additional data.

The Data Delivery subsystem may implement different delivery methods, ranging from local file-based storage to centralized database storage accessible over a network. Each entity may either execute actively, in its own thread, or execute in response to notifications from the Data Storage entities, or event in response to user request.

Since a Data Delivery subsystem is not essential for basic operation of the infrastructure, the design of a full-fledged Data Delivery subsystem is beyond the scope of this thesis, and is left for future work. A basic implementation of the infrastructure could either provide a simple delivery method which would just dump event record data to a file in response to a user request or upon application termination.

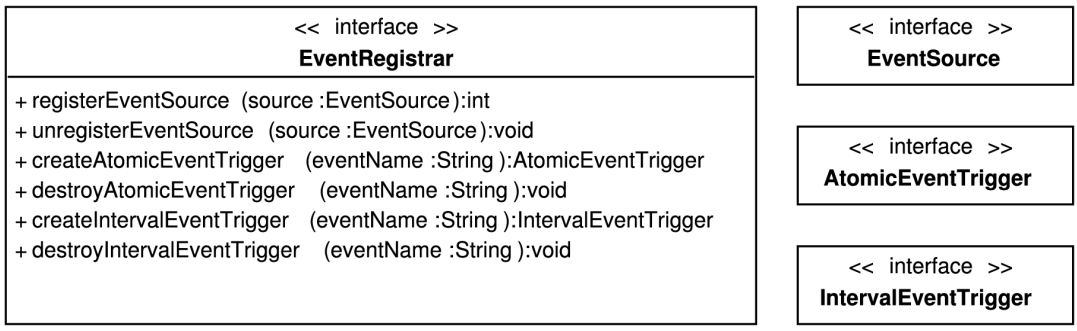


Figure 30: Infrastructure Management interfaces for event sources

Interfaces for Infrastructure Management subsystem

Consequently, the interfaces shown in Figure 29 provide only a sketch of interfaces envisioned to be provided to the Infrastructure Management subsystem. The DataDeliveryManager provides information about delivery methods provided by the subsystem and allows creating a controlling entity for each of the methods. The DeliverMethodDescriptor provides an internal name and description of a particular delivery method, along with a list of properties that represent the method configuration options. After creating a DeliveryController for a particular method, the controller can be configured and associated with multiple performance events. After calling the startDelivery method, the controller is responsible for delivering the data of associated events, for which it will use the EventDataSource interfaces of the Data Storage entities responsible for processing and storing event data in memory.

The Data Storage subsystem could be also implemented externally to the measurement infrastructure which may be, after all, a better option. This would only require designing a defined structure of EventRecord instances so that the data contained within can be understood outside the measurement infrastructure. The EventDataSource interface is mostly prepared for this option, but there would need to be additional synchronization between external Data Delivery entities and the infrastructure to avoid destroying Data Storage entities while they are accessed from outside.

Infrastructure Management

The Infrastructure Management subsystem of the measurement infrastructure is the part which holds all the pieces together and in fact represents a higher-level subsystem, built on top of other subsystems. It is mainly responsible for maintaining the association between performance events and performance data, and providing suitable interfaces for delegating requests to other subsystems that provide functions that need to be provided outside the infrastructure.

Interfaces for event sources

Probably the most important interface provided by the Infrastructure Management subsystem is shown in Figure 30 and concerns registration of event sources with the measurement infrastructure. However, the functionality of the EventRegistrar interface is implemented by the corresponding methods of the EventSourceManager interface provided by the Event Sources subsystem. The Infrastructure Management subsystem is mainly responsible for converting the simple string-based event names to an internal representation of an event name, before delegating the calls to the EventSourceManager

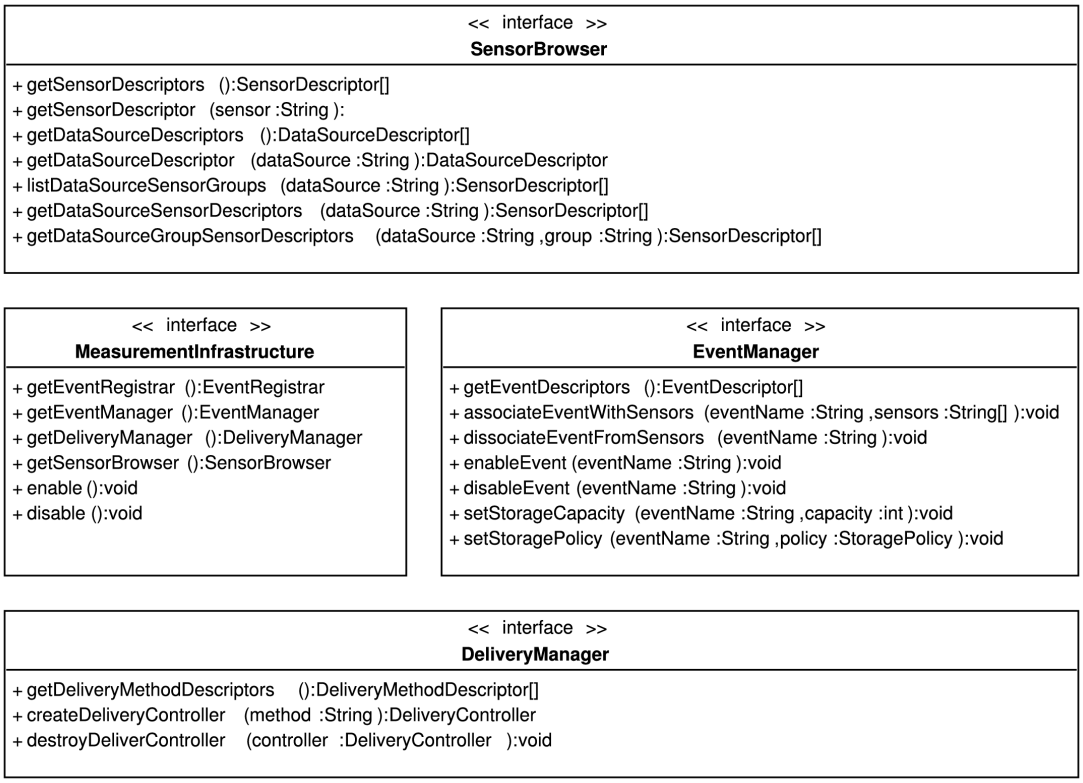


Figure 31: Infrastructure Management interfaces for external management

interface. The EventRegistrar interface also serves to only provide partial, supervised access to the Event Sources subsystem. By delegating the calls, the Infrastructure Management subsystem is automatically notified about new event sources in the system.

Interfaces for external management

Besides registering event sources, the infrastructure should provide additional management interfaces to provide programmatic access to information, configuration, and control of the infrastructure.

Concerning information, the Infrastructure Management should allow querying available performance events, performance data, and data delivery methods. Concerning configuration, the Infrastructure Management has to allow associating events with performance data, configure the storage policy and capacity for a particular event, and configure a particular data delivery method, which includes association with one or more events. And finally concerning control, the Infrastructure Management has to allow enabling and disabling particular events, starting and stopping data delivery, and enabling and disabling the processing of all events at infrastructure level. These functions are provided by interfaces shown in Figure 31.

The MeasurementInfrastructure is the main interface of the infrastructure, which provides access to interfaces related to various tasks. In addition, it includes two methods that can be used to instruct the infrastructure to enable or disable all event sources configured to emit performance events.

The SensorBrowser interface is only informational and allows querying the available performance data in form of sensors. Methods called on this interface are delegated to the

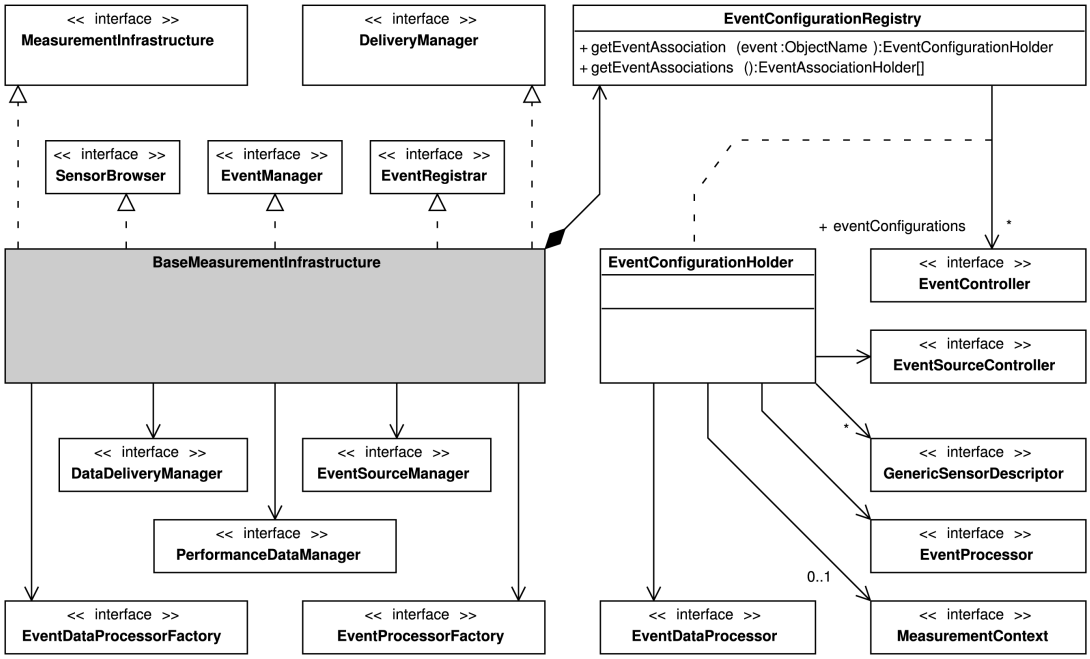


Figure 32: Infrastructure Management subsystem architecture

PerformanceDataManager of the Performance Data subsystem. The EventManager interface provides information about available events, allows associating events with performance data, configuring the storage policy and capacity, and enabling or disabling individual events. The DeliveryManager interface provides information on available delivery methods and allows creating instances of specific methods represented by the DeliveryController interface, which then provides configuration and control capabilities for a particular delivery method instance.

We would like to emphasize that apart from the EventRegistrar and the Measurement-Infrastructure interfaces, the management interfaces shown in Figure 31 are not essential for correct function of the infrastructure. A basic implementation of the infrastructure would be typically configured from file, without the need for programmatic configuration interface. Similar to the Data Delivery subsystem, the management interface is needed mainly for long running production applications, where a user needs to be able to configure the infrastructure at runtime. The above interfaces are therefore provided mainly for illustration purposes and their final design is left for future work, to be decided based on specific needs.

Internal architecture and operation

Since the Infrastructure Management itself does not perform any complex operation, the internal architecture shown in Figure 32 is also rather simple. The BaseMeasurement-Infrastructure class in the left half of the diagram represents the main implementation of the subsystem. The realized interfaces capture the concept of different externally-accessible facets, while the references to interfaces provided by other subsystems capture the spider-like nature of the Infrastructure Management subsystem.

Besides exposing trimmed-down versions of their interfaces and delegating calls to other subsystems, the Infrastructure Management subsystem is responsible for building the event processing pipeline for each enabled performance event. To keep track of the current state, the implementation of the subsystem uses the EventConfigurationRegistry

class, which maintains association between a particular event represented by an EventController, and a number of infrastructure entities that form the event processing pipeline. The concept is represented by the EventConfigurationHolder association class, shown in the right half of the diagram.

Whenever a user enables processing for a particular event or associates an event with a set of sensors from the Performance Data subsystem, the implementation of the Infrastructure Management subsystem performs the following steps:

1. Try to locate the event in EventConfigurationRegistry. If no EventConfigurationHolder has been found for the event, it means that there are no entities related to the event and need to be created first, for which it needs to:
    1. Use the EventSourceManager to locate EventController for the event and use it to obtain a reference to the corresponding EventSourceManager.
    2. Determine the type of the event using the EventController.
    3. Use the EventProcessorFactory and EventDataProcessorFactory to create an EventProcessor and EventDataProcessor corresponding to the event type.
    4. Configure the EventDataProcessor to default values concerning storage capacity and policy and assign it to the EventProcessor.
    5. Assign the EventProcessor to the event using its EventController.
    6. Create a new EventConfigurationHolder instance, fill it with the obtained information and register it in the EventConfigurationRegistry.
2. If a user requested associating an event with performance data, the subsystem:
    1. Uses the PerformanceDataManager to create a MeasurementContext corresponding to the selected sensors.
    2. Provides the MeasurementContext to an EventProcessor as a base measurement context. Any earlier performance data associations are discarded. Depending on the implementation, it may need to first disable the event using the EventController and re-enable it (if it was originally enabled) after setting the base context.
    3. Updates the EventConfigurationHolder to reflect the new association.
3. If a user requested an event to be either enabled or disabled, the subsystem uses the EventController to perform the requested operation. At this moment, the internal structures of the infrastructure should be ready for receiving event notifications.

Additional management considerations

For coordinating the configuration and operation of multiple instances of the measurement infrastructure from a single application, the management interfaces have to be remotely accessible. This could be done using standard middleware platforms such as CORBA or RMI. The types exposed in the interfaces are very too complex, but still the interfaces may need additional work to allow using the middleware conveniently.

However, instead of special remote interfaces, each infrastructure instance can provide a standardized management interface, based either on the SNMP or JMX technologies. In our early proof-of-concept implementation of part of the infrastructure, we have used JMX to configure the infrastructure at runtime, using the `jconsole` tool from SUN Java SDK. This has proved to be sufficient and only required creating management objects for different entities in the infrastructure providing access to information and different operations. We have used the generic system of object names defined by JMX to allow navigation through a hierarchical structure of available events, which provided easy location of event sources in an instrumented application. Since the JMX object names are generic, other entities besides events can be exposed in a different hierarchy of names, all accessible by the generic tool.

Specifically for this purpose, the internal structure of object names corresponding to events and other entities of the infrastructure is identical to the structure of names used in JMX. Even though JMX is Java-based, the object name structure is not platform specific and can be used the same way on different platforms.

Chapter 5

Performance Instrumentation

Application instrumentation is a process of modifying an existing application to provide a particular function that is typically orthogonal to the primary purpose of the application and is therefore unsupported by the application code. To become a part of an application, the instrumentation code implementing a new function needs to be included in the execution flow of an application. This is done by defining and intercepting significant events in application execution, and redirecting the flow of execution at the occurrence of such events to the instrumentation code which then resumes the execution of original application code.

A variety of instrumentation methods exists, each supporting different types of execution events that may be intercepted to include alien code in the execution of an application. The difference in supported types of execution events is partially tied to the form of the application a method is intended to work with. At one end of the spectrum are instrumentation methods based on analysis and modification of application source code, while at the other end there are generic methods based on virtual machine support for interception of typical execution events such as method invocation or creation of an object instance.

For illustration, the concepts behind application instrumentation provide a foundation for aspect oriented software development. AOSD [KH+01] is a programming methodology based on identification of *crosscutting concerns* in an application and separation of code implementing such concerns from the main application code into reusable modules called *aspects*. Besides implementation code, the definition of an aspect also includes the conditions which lead to execution of its code. The basic events which may lead to execution of an aspect are represented by *joinpoints*, which correspond to primitives of execution such as method invocation, or variable assignment. A selection of joinpoints is then represented by one or more *pointcuts* associated with each aspect. The aspects are integrated with the main application code in a process referred to as *weaving*, which applies the aspects in the context defined by their pointcuts.

Aspect weaving is typically performed on application source code, but in general it can use any method to integrate aspect code with application code, such as native code or byte code manipulation. However, source-based aspect weaving is a natural method when using aspects in software development and was originally the only weaving method supported by AspectJ [EF06], the main driving force behind AOSD.

On the other hand, application instrumentation and associated techniques are not intended for application development. Rather, the goal is to augment existing applications in any form with code that originally was not there. The instrumentation code providing a particular function can be either specific to a particular application or generic, which allows using the same code base for different applications. However, generic instrumentation frameworks typically have to generate the instrumentation code. Such application independent instrumentation frameworks are conceptually very close to a

collection of aspects implementing a particular crosscutting concern. With the advent of other than source-based aspect weaving methods, generic instrumentation frameworks utilizing the AOSD infrastructure for purposes typically addressed by instrumentation appeared, such as e.g. Glassbox [GBC06], which is a generic framework for performance evaluation of Java enterprise applications.

This brings us back to the role of application instrumentation in achieving the goals of this thesis. Collecting design-level performance data for heterogeneous component-based applications in a generic way requires modifying the application to provide performance data and deliver it to interested parties, hence the name performance instrumentation. The main responsibility of performance instrumentation is to define performance events [SM06] and to associate those events with collection, processing, and storage of performance data. Technically, the performance instrumentation layer serves as a bridge between code providing performance data collection capabilities and the original application code.

A brief overview of techniques that can be used for application instrumentation has been already presented in Section 3.1.3 during elaboration of requirements related to performance instrumentation. However, those requirements also exclude most of the mentioned techniques from consideration, because they are not generally suitable for instrumentation of component based applications. We will therefore consider only a very limited subset of those techniques for performance instrumentation.

In this chapter we provide a solution to accomplish the objectives of the second subgoal of this thesis. Specifically, we propose a connector-based instrumentation technique to enable non-intrusive, generic, and transparent instrumentation of component-based applications which is necessary to define design-level performance events. We also describe the design of an instrumentation layer which associates performance events with collection of performance data using the generic measurement infrastructure. The design of the instrumentation layer along with the properties of a connector model used for performance instrumentation enables runtime management of execution overhead induced by the instrumentation layer.

Note: Sections 5.4 and 5.5 were taken verbatim from [BB06a] and adapted for use in this thesis. The results published in [BB06b] remain valid and are not influenced by the changes or occasional different wording used to improve the readability of the original text.

5.1 Instrumenting Component-based Applications

The requirement for non-intrusive performance instrumentation is not merely a convenience. Even in case of relatively simple and straightforward component models with no standardized deployment process, such as the CCA platform, classic instrumentation techniques based on modification of application code after being loaded into process address space are of limited utility [SMA01], therefore requiring performance instrumentation to be done using non-intrusive techniques suitable for component-based environment. More so in the context of heterogeneous component-based applications that are launched using a complex deployment process, and for which the source code is not available.

5.1.1 Choosing an instrumentation technique

When choosing an instrumentation technique for an application, there are several aspects that need to be considered:

- what is the purpose of the instrumentation code and how it will be obtained,
- where does the instrumentation code belong in the execution flow of an application and how to intercept and redirect the flow to include the instrumentation code, and
- what moment of an application life cycle is suitable for performing the instrumentation.

In practice, all these aspects are mutually interconnected and the order of their resolution depends on the overall requirements imposed on the instrumentation process, because together, they determine the level of abstraction at which execution related events can be defined and the transparency of the instrumentation process.

In our particular case, the choice of an instrumentation technique was the last aspect to consider, because the purpose and origin of the instrumentation code as well as the application life cycle stage for performing the instrumentation have been, to a large degree, determined by the requirements on the instrumentation process.

Instrumentation code

Generally, the instrumentation code can be written manually or generated, either in source or binary form. However, since the instrumentation method must be non-intrusive and must not require developer assistance, the performance instrumentation code has to be automatically generated. This in turn requires the instrumentation code to only perform essential tasks to allow keeping the code generating the instrumentation code simple and easily adaptable to support different programming languages.

Instrumentation code responsibilities

One of the main principles behind the design of the measurement infrastructure presented in the previous chapter has been to require only minimal instrumentation code to serve as a bridge between the measurement infrastructure providing performance data collection capabilities and the original application. Consequently, the instrumentation code is responsible for two tasks. The first task is to initialize and configure the measurement infrastructure. This is done by providing the measurement infrastructure with information about performance events, provided by a particular instance of the instrumentation code, which can be associated with collection of performance data. The second task is to notify the measurement infrastructure about the events when they occur.

These requirements make the instrumentation code that needs to be generated rather simple, but also very mutable, because it needs to be adapted for each occurrence in the application. If not for the goals of this thesis, this makes the manual production of the performance instrumentation code infeasible, even though it may be supported for testing and debugging purposes in small scale application. Moreover, since component-based applications typically provide explicit description of their architecture, including connections among components and interface types used in these connections, generating the performance instrumentation code is only natural, because the necessary information is available to a generator.

Instrumentation timing

Application instrumentation can be performed basically at any stage of application life cycle, assuming there is anything to instrument, i.e. an application must be at least in

development phase. At one end of the spectrum of instrumentation opportunities is manual instrumentation of application source code, while the ideal solution at the opposite end of the spectrum would entail connecting to a running application, selecting entities that should be instrumented, and performing the instrumentation at runtime only on the selected entities. However, such a solution would depend on many specifics of a particular execution platform, and therefore is not a suitable candidate for a generic instrumentation method for heterogeneous component-based applications.

Instrumentation at deployment time

To comply with the requirements set forth in this thesis, the performance instrumentation process must be integrated with application deployment. However, besides that specific requirement, there are other requirements that make deployment a suitable life cycle stage for instrumentation. Due to requirement on non-intrusiveness of the instrumentation process, we have to assume that the application source code is unavailable as opposed to implementation artifacts and metadata describing the application architecture. Since the process has to be generic, it has to operate on a standardized form of an application, which is typically defined at the level of application packaging entering the deployment process. This leaves us with the options of performing a pre-deployment instrumentation on an application package, which is done e.g. by COMPAS [Mos04], instrumenting an application during the deployment process, or performing a post-deployment instrumentation, which is also an option provided by COMPAS.

Since performance instrumentation involves generation of instrumentation code based on application metadata and may even require information on assignment of components to computational nodes, the deployment process is a suitable candidate for integration of performance instrumentation because it provides standardized access to application metadata as well as additional information related to physical deployment of application components. Moreover, the deployment process defined in the OMG specification [OMG06c] already requires an application deployer to use tools which provide opportunities for integration of the performance instrumentation process, thus allowing to achieve transparency with respect to an application deployer.

Instrumentation technique

The choice of an instrumentation technique determines the basic level of abstraction at which execution related events can be defined and intercepted and thus ensure integration of instrumentation code with the original execution flow of an application.

Intrusive techniques

Intrusive instrumentation techniques have an undeniable advantage in that they modify the internals of application components and leave the application architecture intact. Such techniques are therefore are completely transparent with respect to the runtime of a particular component system. As a result, there are no problems with e.g. component reference passing, because references point at real components and there is no need to intercept reference passing among components. Features such as dynamic update or architectural reflection are also unimpeded by the instrumentation. However, intrusive instrumentation technique either require access to application source code or need to be performed using native code or byte code manipulation, which makes them unsuitable for general application in the context of heterogeneous component-based applications, because they are very specific to a particular execution environment.

Non-intrusive techniques

Of the plethora of application instrumentation techniques, only a few are generally suitable for heterogeneous component-based applications. These techniques are non-intrusive and leave the internals of application components intact. For integration into application execution, they exploit the compositional approach to construction of component-based applications. However, they also often require the application architecture to be modified, because each component is impersonated by one or more additional entities which contain the instrumentation code and delegate requests made on them to the original component. Such delegation imposes only minimal requirements on an execution platform as it only needs to support indirect method invocations.

The impersonation is typically achieved through wrappers or proxies. Wrappers encapsulate a target component with all its external interfaces and maintain a facade compatible with the target component. Calls on the interfaces exposed by the facade of a wrapper are delegated to the external interfaces of the encapsulated component. Proxies work in a similar way, except there is typically one proxy object for each external interface of the original component and proxy objects are not responsible for the life cycle of the original component.

Problems with wrappers and proxies

Because of the additional entities required to impersonate components, there are additional problems that need to be addressed when using wrapper and proxy-based instrumentation techniques, especially in distributed environment.

Most of the problems arise from the need to address the life cycle of the runtime entities representing wrappers and proxies. Typically, proxies and wrappers are represented as normal components, which solves the problem of their life cycle, but for that they have to be explicitly include in the application architecture. Even though the modification of the architecture can be done automatically, problems may arise from different semantics of connections among the original application components and connections between a wrapper or a proxy and the impersonated component.

In distributed environment, components are often using communication middleware which itself uses proxies to achieve distribution. However, these proxies are typically part of the implementation of application components. Using proxies for instrumentation would require moving the middleware proxies from the implementation of the original components to the proxy components, and maintain a strictly local binding between a proxy and its target. If we want to support proxies for both provided and required component services to e.g. provide client context to the instrumentation present in a server-bound proxy, we may need to change the interface in proxy-to-proxy communication to pass the additional information. However, this would require to completely regenerate the middleware proxies to reflect the modified interface.

Problems with life cycle and middleware can be avoided if the wrappers and proxies can be made part of a component representation at runtime without the need to explicitly include them in application architecture. This can be achieved in specific context of a particular component system. The COMPAS [Mos04] framework uses inheritance-based wrappers, which replace the original components in application architecture and the access to their implementation is maintained implicitly through access to a superclass. In case of the Julia [OWC07d] implementation of the Fractal [OWC07a, BC+06] component model, the concept of a proxy is directly supported by the component runtime, because the runtime representation of a component consists of multiple entities, with separate entities for component interfaces and their implementation. The entities representing

component interfaces explicitly support interceptors, which therefore become part of the component runtime. Similar result can be achieved by middleware specific interceptors, such as the portable interceptors found in CORBA.

While integrating proxies with a particular component runtime provides an effective solution to the mentioned problems, it still does not allow changing the interface in proxy-to-proxy communication. Moreover, such solutions are very specific to a particular component and middleware technology. Despite the disadvantages, the basic concept of impersonation and delegation found in wrappers and proxies is still generally suitable for heterogeneous component-based application, because it does not require application source code, is non-intrusive, and has very little requirements on the underlying execution platform.

As a result, to utilize the basic concept of a proxy and to avoid the problems related to using wrappers and proxies, we have decided embed the performance instrumentation code in connectors. Connectors are entities found in several component models to mediate communication among components. They are very similar to proxies, however, they are an inherent part of application architecture, the binding between a connector and a component has explicit semantics, and at runtime they encapsulate middleware code implementing distributed communication. Consequently, connectors are not susceptible to the problems plaguing the use wrappers and proxies in component-based applications.

5.1.2 Connector-based application instrumentation

Software connectors are first-class entities used in component systems to model and realize communication among components. In such systems, components are intended to only contain application logic, while connectors are responsible for realizing the communication among components participating in a connection using a prescribed communication style. This also includes communication in a distributed environment spanning multiple address spaces, for which the connectors typically utilize existing middleware.

At the design level, connectors are used to model the requirements on the connections they represent. In application architecture, connectors are represented by edges connecting two or more components, with each of the edges associated with a set of properties capturing the requirements on the respective connection. These requirements then serve as a high-level specification which determines the runtime representation of a connector.

To effectively realize distributed communication, the runtime representation of a connector is inherently a distributed entity. As shown in Figure 33, a connector consists of a number of connector units, each representing part of a connector that can independently exist in an execution environment (deployment dock), which hosts both components and connectors. Each connector unit is responsible for local communication with a component attached to it in the deployment dock, and for remote communication with other connector units (typically using middleware) that together form a connector.

At runtime, connectors are similar to proxies and wrappers in that they are bound to the interfaces through which a component communicates, but unlike proxies and wrappers in general, they are part of the application architecture, both at design time and at runtime. The runtime integration is especially important because it provides connectors with explicit life cycle management, the lack of which was the main source of problems associated with wrappers and proxies.

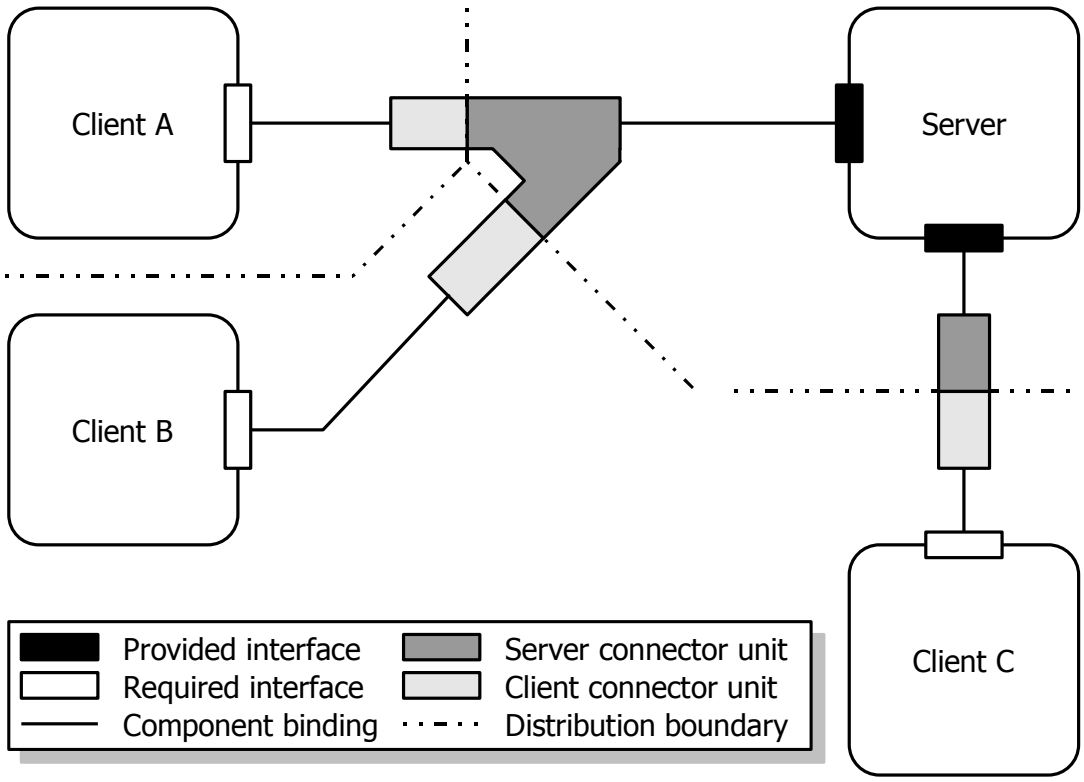


Figure 33: Using connectors to mediate communication among components

The implementation of connectors can be generated after an application has been assembled, which provides a major benefit in design and development of component-based applications. Using connectors in application design allows an application developer to postpone decisions on issues such as what communication middleware will be used to address distribution. However, this does not make distribution just a side effect that does not need to be considered in application design; quite the contrary. A developer still needs to consider address space boundaries between components because they are part of the architecture. Moreover, a developer also needs to be aware of the fact that using middleware to address distribution in heterogeneous environment restricts the set of types that can be exposed in component interfaces intended for communication across address spaces. At deployment time, the implementation of connectors can be generated to employ different technologies to implement distributions depending on their availability on the target platform.

The explicit life cycle management and the fact that they can be generated makes connectors extremely suitable for placement of performance instrumentation code. The application architecture remains intact, there are no issues with placement of middleware specific proxies, and because connectors are associated with components, they are a natural place for definition of design-level performance events in component-based applications. Since connectors will be explicitly managed by a component system runtime, the performance instrumentation code will be automatically put in the right place without the need to know the details of runtime representation of components in a given component system.

Connectors are generated by a connector generator, which is responsible for producing a connector implementation which satisfies the requirements associated with a particular connection represented by a connector. To choose a suitable middleware for distributed

communication, the generator requires information concerning assignment of application components to computational nodes as well as information concerning middleware platforms supported by individual nodes. This kind of information is available during application deployment, which is therefore the most suitable moment for generating connectors.

Making the instrumentation code part of connector implementation allows generating it during application deployment along with other parts of connector implementation. The integration with connectors makes the performance instrumentation process fully automatic and completely transparent. This is consistent with the requirements on the performance instrumentation process.

There are several component systems with connector support, but the above solution can only be applied in component models with explicit connector support both at design time and runtime. A comparison of component models with respect to connector support presented in [Bur06] shows that such level of connector support is not yet very common.

In the following sections we provide an overview of a connector model with a number of distinguishing features that are not commonly present in other connector models. The model is based on a compositional approach to construction of connectors and preserves design-time architecture of a connector at runtime. We then describe how to use this particular connector model to achieve generic and non-intrusive performance instrumentation of component applications.

5.2 Overview of Architecture-based Connectors

Since its inception, the model of architecture-based connectors [BP01, Bal02], which we use for the implementation of performance instrumentation in connectors, has undergone several evolutionary changes. The changes were prompted by the need to resolve various ambiguities and to support generation of connector implementation [BB03], different communication styles [BP03, BP04], and deployment of heterogeneous component-based applications [GB05a, GB05b].

In this section we aim to introduce the key concepts and features of the connector model related to the work presented in this chapter. Detailed description of the connector model including comparison and relation to other connector models can be found in [Bur06].

5.2.1 Connector meta-model

Technically, at runtime there is little difference between regular application components and connectors. However, compared to application components, connectors have a specific purpose in mediating the communication of other components. Even though the implementation of a connector is specific to a particular application, the scope of operations and concepts employed by connectors is somewhat limited.

This specialization of connectors allows reusing communication related code with finer granularity and at different level of abstraction than in case of application components. The reuse takes place at the level of middleware, which has been successfully abstracted away from applications.

The meta-model of architecture-based connectors promotes compositional approach to construction of connectors, which is based on combining a number of lower-level entities implementing various middleware-related concepts into higher-level entities that mediate communication among components using a particular communication style – connectors.

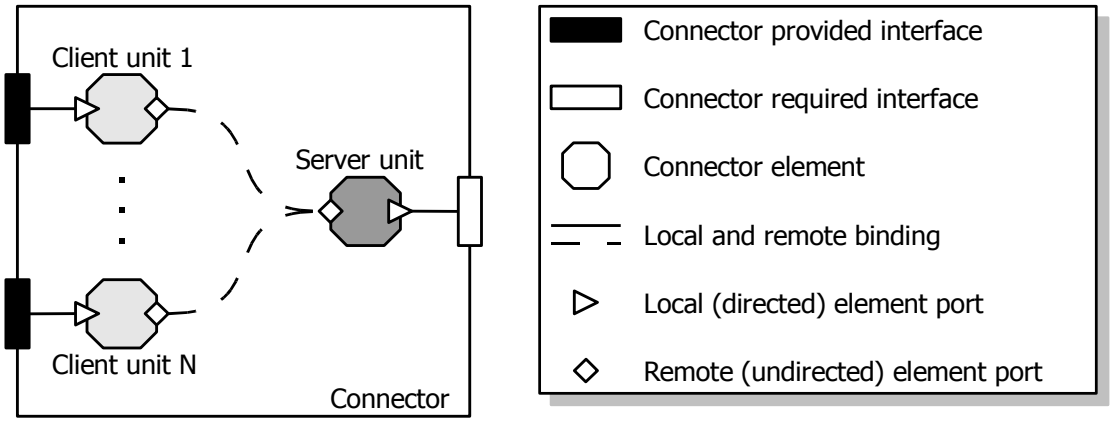


Figure 34: Architecture of a connector

Connector architecture

The architecture of a connector is a hierarchical structure, which represents the composition of elements implementing the concepts necessary for realization of a given communication style. At the top level, a connector is a single entity connected to all components participating in the communication. At this level, a connector can be viewed as a black box exposing *local ports* that can be bound to complementary ports with the same signature exposed by components.

Since a connector is an inherently distributed entity, its implementation must have explicitly defined distribution boundaries, because the internal architecture of a connector depends, among other things, on the way it is distributed. The purpose of the second level of the connector architecture is to provide this kind of information. There, a connector is split into several *connector units* associated with components bound to the local ports exposed by a connector. The first two levels of connector architecture are shown in Figure 34.

With the information available at the level of connector units, a connector still resembles a black box, or more precisely, several black boxes. Connector units define the ports that are exposed at the top level of connector architecture, and through which a connector attaches to the application components. Connector units also define *remote ports*, which are used exclusively for communication among connector units. The architecture of a connector at the level of connector units and connections among them reflects a communication style realized by a connector.

The next level of connector architecture describes the internal architecture of connector units. Each connector unit is a composition of mutually interconnected *connector elements*. Connector elements define the ports that are exposed at the level of connector units and also carry the implementation code of a connector.

Connector elements

Connector elements function as containers for implementation of concepts that make up a connector and are thus its basic building blocks. While a connector is a distributed entity spanning multiple address spaces, connector element is a local entity restricted to a single address space. Connector elements can be nested, which allows expressing complex concepts as a composition of simpler concepts.

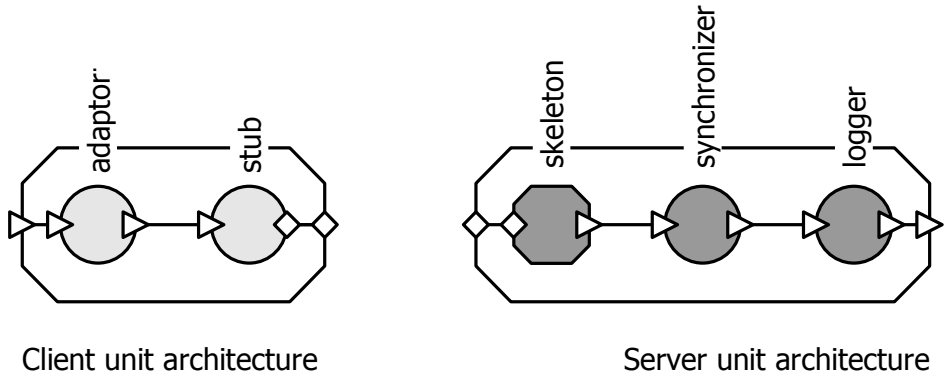


Figure 35: Architecture of connector elements

To allow composition, connector elements define local and remote ports through which they can be connected to components and other connector elements. Together, the ports defined by an element represent a *connector element type.*

Implementation of a connector element type is provided by a *connector element architecture*. These two concepts play the role of a black-box and a gray-box view of a connector element. The separation of connector element type from its architecture allows providing multiple implementations of a particular element type. Consequently, there can be multiple implementations of a particular connector unit and a connector.

The architecture of a connector element can be either *primitive*, or *composite*. Primitive element architecture represents a leaf in a connector architecture hierarchy and is backed by a code template implementing a particular connector element type. Composite element architecture, on the other hand, represents an internal node of the connector architecture. In addition to code template implementing the element, a composite architecture also contains instances of other element types and describes connections between them as well as between the child elements and the parent element.

Using element types in specification of child element instances increases the flexibility of the entire connector architecture with respect to implementation, because each element type can be implemented by multiple connector architectures.

Technically, a connector unit is a connector element as well; therefore the minimal depth of the entire connector architecture hierarchy is two. However, for practical usability of the concept of architecture-based connectors, the connector architecture hierarchy will typically have at least three levels, which allows composition and reuse of basic middleware-related concepts.

Figure 35 provides a detailed architecture view of the two composite connector elements representing the connector units originally shown in Figure 34. Each composite element only specifies instances of element types and does not define what architecture should implement the element types it instantiates. Therefore each of the elements contained in the two architectures can again be a composite element, however in this particular case most of the elements will be implemented by a primitive architecture.

Connector element ports

Connector element ports provide endpoints for communication among connector elements in a connector as well as among connector elements and components bound to a connector. Ports contain interface signatures or signature templates along with additional

information not included in interface type definition. As mentioned above, we distinguish between local and remote ports depending on the type of connections the ports are allowed to participate in.

Local ports are intended for local connections, which serve for communication between connector elements within a single connector unit and for communication between connector units and components. Local communication is realized by procedure calls and takes place within the confines of a single address space. Connection between local ports is therefore directed and we need to distinguish the caller and the callee. For this purpose there are two kinds of local ports, which represent the two roles in procedural communication.

The first kind is *provided port*, which provides an interface reference at runtime, and therefore represents the callee. The other kind is *required port*, which requires an interface reference at runtime, and therefore represents the caller. Both kinds of ports contain a single interface signature or signature template. All element types in architectures depicted in Figure 35 define at least one local element, which allows them to be bound to other elements in the same address space. Element types such as adaptor, synchronizer, or logger only define local ports and therefore cannot represent a connector unit element.

Remote ports are intended for remote connections, which serve exclusively for communication among connector units within a connector. The realization of remote communication is an issue of implementation, which depends on the capabilities of the execution environment hosting an application. Typically, remote connection will be realized by existing middleware implementing a standard associated with a particular communication style, e.g. in case of procedure call communication style these would be the RMI or CORBA standards.

Even though in case of procedure call communication style the direction of the connection can be identified and has a clear meaning, in message or blackboard based communication styles this is not the case. Moreover, remote connection can be expressed as hyper-edge to model other communication mechanisms, such as broadcast and multicast.

Remote connection is therefore considered undirected and the connection is established at runtime in an implementation specific way. Remote ports are only used to assist the implementation of remote connection in exchange of references necessary to establish the connection.

Consequently, *remote port* is the only kind port for remote connections and the third kind of port in the connector model. In contrast to local provided and required ports, remote ports may contain multiple interface signatures or signature templates. The architectures of the two connector units shown in Figure 35 contain a primitive element with a single remote port which is used to connect the two connector units. The server unit contains a skeleton element, while the client unit contains a stub element. In practice, these elements would ensure the exchange of references needed to establish connection between a stub and skeleton in the underlying middleware technology, such as CORBA.

Connection restrictions

All connections in the entire connector architecture hierarchy are subject to certain restrictions. Connecting two local ports of complementary kinds with the same interface signature results in a *local binding*, which is only allowed between sibling connector elements, with the exception of connector units. Connecting two ports of the same kind

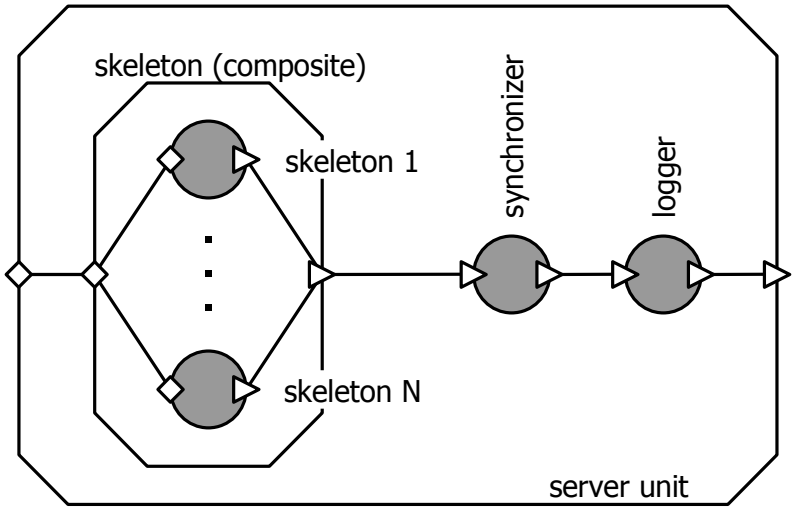


Figure 36: Server connector unit with multiple skeletons

with the same interface signature results in a *local delegation* and is only allowed between parent and child connector entities. Other local connections are not allowed.

Since there are no complementary remote ports, *remote binding* is a connection among remote ports of sibling connector elements, which is only allowed at the level of connector units. *Remote delegation* is a connection between two remote ports of parent and child connector elements.

The result of these restrictions is that a connector can only expose local ports delegated to and from connector units. Remote bindings can only exist among connector units within a connector. If there is a child connector element with a remote port, the port must be delegated to a remote port of the parent element, possibly recursively, until it reaches the element representing a connector unit.

Connector architecture may contain multiple instances of connector element types, including connector units. The number of instances is determined by a cardinality of an element, which can be either *one* or *multiple*. This can be used to model connector architecture with a single server unit and multiple client units, or to specify architecture of a connector unit with multiple stub or skeleton elements, which allows using a single connector unit with different types of middleware. Architecture of a server unit supporting multiple skeletons is shown in Figure 36.

Port signature templates

As mentioned earlier, connector elements and their ports are basically templates to allow adaptation of connector types to different interfaces. The interface signatures in element ports are therefore only symbolic, with a particular component interface type replacing the symbolic signature during connector configuration which is described later.

In many cases, an element needs to transform the original interface exposed by a component. In case of local ports, this transformation may include adding or removing interface methods or adding or removing method parameters. Such a transformation is necessary e.g. to add caller identification to method calls performed on a server component. This would be achieved by a pair of connector elements, one of them in a server connector unit and the other in a client connector unit. The element in the client unit would expose a local provided port compatible with the component required

interface, and a local required port featuring an interface containing methods with an additional parameter representing the caller identification. During execution, this element would add the caller identification to invocations of methods occurring on its provided port. In symmetry, the element in the server connector unit would expose a local provided port featuring the extended interface and a local required port compatible with the server component provided interface. During execution, this element would strip the caller identification from method invocations on its provided port and delegate method invocations using the original interface to the server component (or a next element in chain).

Because the architecture specification does not operate on concrete interfaces, these transformations are described as functions applied to the original symbolic port signature. This allows ensuring signature compatibility between ports of connector elements and during architecture resolution, it allows finding compatible connector elements by matching port signatures.

This approach is also heavily used in stub and skeleton elements, because they often need to provide a remote port with a signature specific to a particular middleware, such as RMI, which is based on the signature of the component interface, but represents a different type. In this context, the functions are also used to express interface semantics. Another application of this approach is to describe signatures of elements performing interface adaptation.

Connector configuration

The connector architecture hierarchy describes, in a very flexible way, only a model of a connector. The interleaving of black-box and gray-box concepts provides the model with a number of variation points, since for every black-box (element type), there can be multiple gray-boxes (element architectures) describing another composition of black-boxes. To actually create a connector instance from the model, all the choices must be eliminated. From thus fixed connector architecture the implementation of a connector can be generated.

Eliminating the choices means recursively selecting suitable architecture to every composite connector element, eventually reaching the leaf primitive elements. The resulting connector architecture will not be ambiguous anymore, because at each level of the hierarchy, element types will be resolved to concrete implementations. Also the template signatures of element ports will reflect the interfaces of components bound to ports exposed at the top level of the connector architecture.

Such resolved architecture is called *connector configuration* and the resolution has to be done for every connector instance in the application. Since this is hardly a task to be performed manually, the resolution is done by architecture resolver which is a part of connector generator. The architecture resolver takes into account the requirements and non-functional properties specified for each connection during application design as well as information about the capabilities of a computational nodes and the execution environments (deployment docks) that are expected to host the individual units of a connector. Again, more details concerning the resolution algorithm can be found in [Bur06].

Non-functional properties

The description of each connector or element architecture is associated with a declaration of non-functional properties satisfied by the architecture. The mechanism is also used to

declare a communication style implemented by particular connector architecture. The specification non-functional properties aids in the architecture resolution process, because it helps pruning the search space of virtually all possible connector configurations, and allows choosing a connector architecture reflecting the connection requirements from multiple, syntactically valid (no conflicts in port signatures), architectures.

The declaration associates a key with a value. In many cases, the value is atomic, but in general it can be a Prolog-like term. This is useful in case of composite architectures, where the value for a particular key depends on child elements present in the architecture. The composite element may then construct the value as a term which propagates the value of a key defined by a child element.

The specification of non-functional property requirements is complementary to the non-functional property declarations in that it consists of a set of predicates which require specific keys to match a specific value, which again can be a term.

5.2.2 Connector runtime

The main reason for basing the approach proposed in this thesis on the connector model presented so far is that the resulting connector configuration is meant to be directly reflected by runtime entities implementing a particular connector instance. The architecture of a connector at runtime thus corresponds to the architecture prescribed by the connector configuration.

Connector runtime is a framework which covers all the aspects related to connectors at runtime. This framework consists of multiple parts with different level of dependency on the environment. The part boundaries can also serve as vendor boundaries, because each of the parts is responsible for a certain aspect of the connector runtime. These aspects are mutually independent; therefore parts of the runtime can be produced independently by multiple vendors.

The connector runtime comprises the following aspects: implementation of connector elements, connector management and runtime model, and integration with a particular component model.

The connector elements make up the implementation of a connector instance. For efficiency, they should be implemented in the programming language used form implementation of components bound to connector units. They are meant to be generated and typically do not depend on a particular component system. In case of elements implementing a remote communication, the generated code may mostly serve to configure third party middleware.

Connector management is responsible for connector instantiation, coordination and exchange of remote references among connector units, and other tasks related to connector runtime support. This part of connector runtime does not depend on a particular component and can be implemented in a single language with bindings to other languages so that connectors and connector units can be accessed by the management code.

Integration of connector runtime with a particular component system provides connector management with the means to access component management interfaces so that connector units can be bound to components. It also ensures that in places where component references are expected, connectors are used instead. This may require modification of the component system runtime, but it ensures location transparency with

respect to the internals of the component system in case of e.g. composite components which may be deployed on different node than the components it contains.

To avoid implementing the connector runtime for each component system from scratch, the model of the connector runtime follows the MDA approach [OMG01] and is based on a platform independent model which provides a unified view of the connector runtime [Bur06]. This model is then transformed into platform specific models for individual component systems.

Platform independent model

The work presented in this thesis is based on the platform independent model of the connector runtime. The platform specific models for individual component systems are not relevant, because the presented approach only applies to connector elements and connector management, neither of which depends on a particular component system.

The platform independent model describes the basic entities, the process of connector unit instantiation and linking, and reference passing. Of these, the presented approach requires understanding of the basic runtime entities, which include the structural elements that represent the connector architecture at runtime. The remaining parts of the platform independent model are mostly relevant to implementation of a connector runtime and will not be reviewed here. Detailed description of the entire platform independent model can be found in [Bur06].

Connector managers

At runtime, the responsibility for connector life-cycle and subsequently the bindings among connector units is spread among several connector managers which form a two-level hierarchy. In each application, there is a single instance of a *Global Connector Manager* (GCM), which is at the top of the hierarchy. The GCM is responsible for application-wide connector management and is used by a number of *Dock Connector Manager* (DCM) instances. Each deployment dock (address space) where connector units are deployed along with application components has a single instance of DCM, which is responsible for connector management within the confines of a deployment dock.

To create an instance of a connector in an application, two basic steps need to be performed. First, the connector units that make up the connector have to be created in the same docks as are their respective application components. Second, when all units have been created, bindings among their remote ports can be established. During connector creation, the GCM plays the role of a server notified by individual DCMs on creation of a connector unit. During the binding of connector units, the GCM plays the role of a client, querying and providing back references to remote ports of connector units managed by the DCMs.

Figure 37 shows the principal interfaces of the Global and Dock Connector Managers. The interfaces contain methods for life-cycle management, binding of connector units, and several auxiliary methods. Their usage will be briefly explained on the process of connector unit creation and connector unit binding.

Connector unit creation

During application launch, the initial architecture of an application is created. During execution, the initial architecture of an application may evolve if the respective component model supports adding and removing components and bindings among them at runtime.

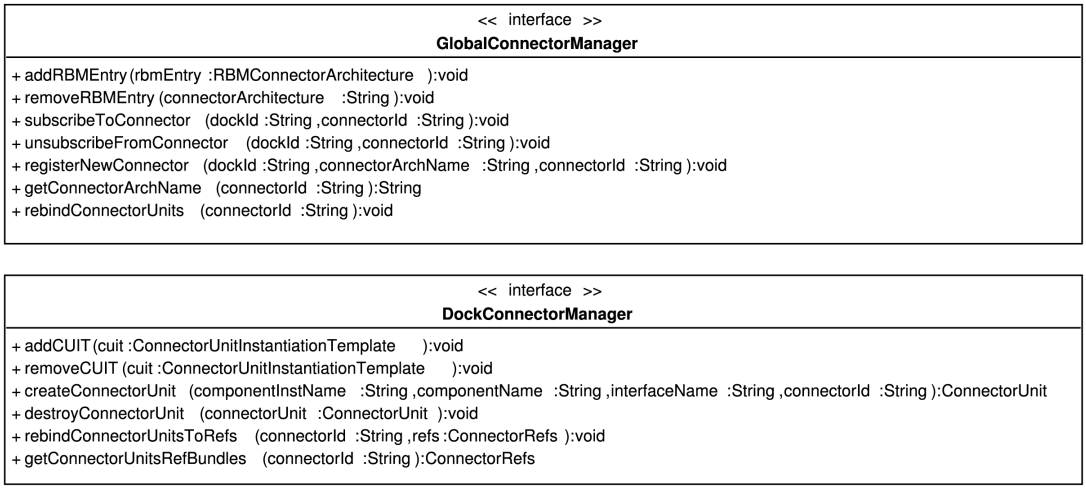


Figure 37: Interfaces of the Global and Dock Connector Managers

These are the two situations during which connector units need to be created. The creation of connector units is initiated from outside of the connector runtime and is specific to the deployment platform.

Since connector units always reside in deployment docks, the responsibility for their life-cycle lies with the DCMs. To create a connector unit, the runtime has to call the createConnectorUnit method on the DCM. After creating an instance of the requested unit, the DCM registers it with the GCM. This is done by calling the registerNewConnector method on the GCM if the newly created unit is the first unit of a particular connector in the dock; otherwise the subscribeToConnector method is called for subsequent units. The list of methods for management of connector and connector unit life-cycle is supplemented by complementary methods destroyConnectorUnit and unsubscribeFromConnector.

When creating an instance of a connector unit belonging to a particular connector, the DCM needs to know what implementation artifacts to use. This information is provided by *Connector Unit Instantiation Templates* (CUIT). Instantiation template associates an implementation artifact with a specific connector architecture and unit. Additional filters associated with an instantiation template can limit its scope to a particular deployment dock, component, component instance, or interface.

Figure 38 shows a class diagram of a Connector Unit Instantiation Template. The instances of the template are stored in a simple repository managed by the DCM. The information held in the repository is provided from outside, by the deployment platform, using the methods addCUIT and removeCUIT methods of the DCM interface.

Connector unit binding

When all units of a connector have been instantiated and registered with the GCM, the remote ports of the units must be bound together to establish communication and thus enable the connector to function. To create a binding among remote ports of connector units, references to these ports must be exchanged among connector units participating in the binding. As in the case of connector unit creation, the process is initiated from outside of the connector runtime, by calling the rebindConnectorUnits method on the GCM for a particular connector.

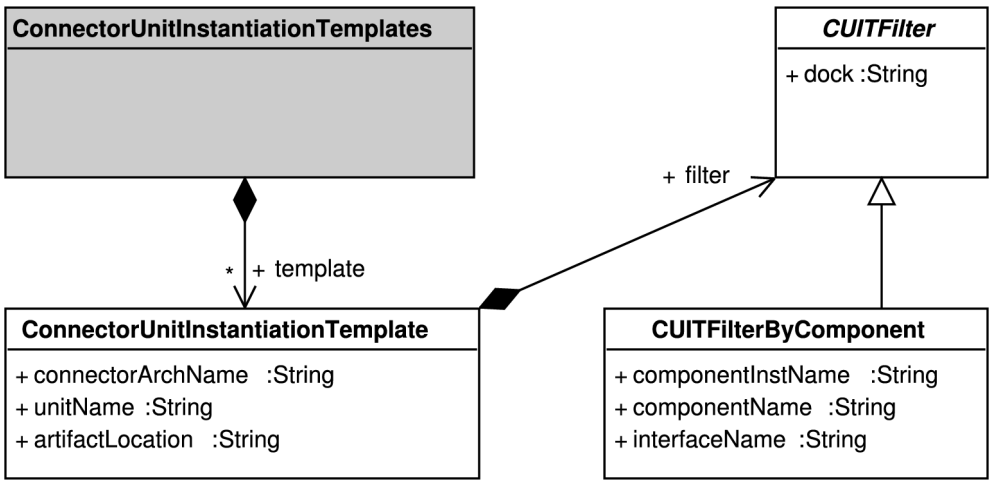


Figure 38: Connector unit instantiation templates

For all connector units registered in the particular connector instance, the GCM queries their respective DCMs and collects from them references to remote ports offering server functionality. This is done by calling the getConnectorUnitsRefBundles method for a particular connector on the DCMs. When the remote port references from all connector units have been obtained, the GCM groups the references according to bindings among the remote ports and provides the references to connector units with remote client ports. This is achieved by calling the rebindConnectorUnitsToRefs method on the respective DCMs.

When grouping the remote references before providing them back to connector units, the GCM needs to know how to distribute the references among connector units. This additional information is contained in *Remote Binding Map* (RBM) which associates connector architecture with a structure describing the bindings among remote ports.

Figure 39 shows a class diagram of the Remote Binding Map. The entries for individual connector architectures are stored in a simple repository managed by the GCM. As in the case of instantiation templates, the information is provided from outside, by the deployment platform, using the addRBMEntry and removeRBMEntry methods provided by the GCM interface.

Connector elements

At runtime, each connector element is represented by a primary class which allows the element to be recognized and manipulated as an architectural element. Depending on the types of declared ports, the primary class has to implement the following interfaces: ElementLocalClient (if an element has a required port), ElementLocalServer (if an element has a provided port), ElementRemoteClient (if an element has a remote port and acts as a client in a remote connection), and ElementRemoteServer (if an element has a remote port and acts as a server in a remote connection). The primary class aggregates the control interfaces that can be used for querying references to element ports (server interfaces) and for binding ports to target references (client interfaces). The signatures of the control interfaces are shown in Figure 40.

Besides the primary class, which is mandatory, the implementation of a connector element will typically consist of other classes, according to the best practice recommended for a given implementation platform or programming language.

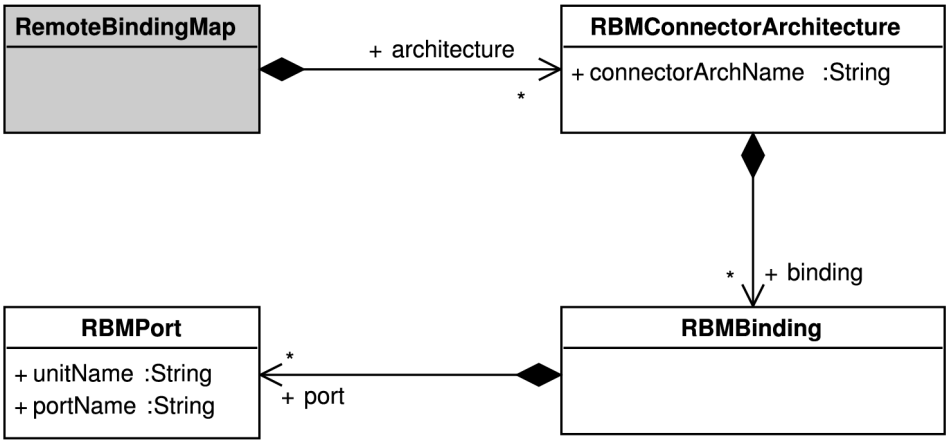


Figure 39: Remote binding map

5.2.3 Connector generator

Generating implementation code of architecture-based connectors is a process of bridging a semantic gap between a high-level human-oriented connector specification, and low-level code implementing a connector. This process has two phases. The purpose of the first phase is to resolve an instance of connector architecture (connector configuration) based on the high-level specification. The purpose of the second phase is to generate code for the elements in the resolved connector architecture, taking into account concrete interface types and specifics of a target execution platform, which includes a particular component system and a programming language.

Architecture resolution has been already briefly described in relation to obtaining a connector configuration in the context of the connector meta-model. When searching for an appropriate connector configuration, the architecture resolver evaluates each connector architecture and the following conditions must hold for architecture to be considered valid [Bur06]:

1. High-level specification consistency, which ensures that architecture satisfies the functional and non-functional requirements of the high-level specification. Verification of high-level specification consistency is related to the NFP evaluation process.
2. Deployment consistency, which ensures that architecture is valid with respect to capabilities of the computational nodes intended for hosting the instances of connector units.
3. Cooperation consistency, which ensures that architecture is syntactically valid with respect to port signatures of all elements. This is related to port signature templates capturing semantic and syntactic transformations on port signatures.

Element generation is the other key part of a connector generator which operates on connector configurations provided by an architecture resolver. Since a connector unit is represented by a connector element, there is no difference between the for the element generator. The main responsibility of the element generator is to generate implementation code reflecting a connector configuration in the runtime of a particular component system, using a particular programming language (typically native to the component system).

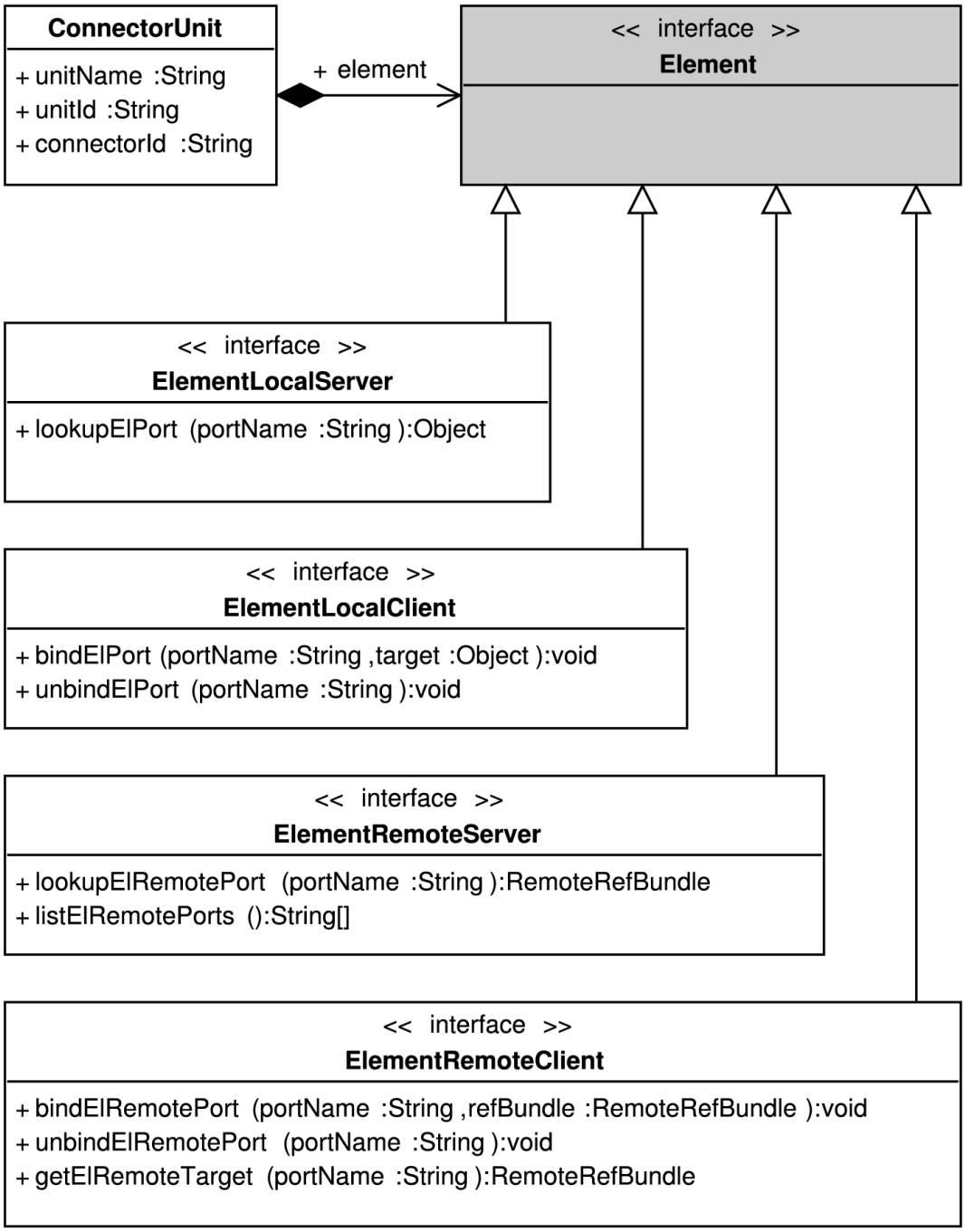


Figure 40: Element control interfaces

The implementation code is based on element templates, which correspond to the entities of the connector meta-model in a particular programming language. These templates contain macros that are expanded to reflect the final port signatures and specifics of a particular component system.

The most important part of the element generator is related to handling types. Since the port signatures are templates parameterized by a specific type, the generator must be able to derive the final signatures of element ports in the type system of the target platform, which includes programming language, component system, middleware, etc. The element

generator therefore operates with an abstract model of a type system, into which it plugs type factories for the type systems it has to work with. The type factories are then responsible for producing types for the respective type system by applying the transformations on port signatures captured in port signature templates. The system of signature templates also allows capturing the need to map types between different type systems. Individual type factories are then responsible for mapping types from other type systems into their native type system.

To enable using existing middleware to implement distributed communication, the generator must be able to use the tools native to a particular middleware, such as RMI or IDL compiler, to generate implementation of elements responsible for remote communication. This is again achieved by a system of plugins which allows adding support for various actions to the element generator. Each connector element described by a script which orchestrates the execution of such actions, including the generation of element code. Further technical details concerning the element generator can be found in [Bur06].

5.3 Performance Measurement Connector

The compositional model of architecture-based connectors allows combining connector features using multiple connector elements. The encapsulation of features in elements turns them into units of work that can be easily reused. In this context, performance instrumentation is basically a connector another feature and as such it must be encapsulated in a connector element.

Since the connector generator employs vertical instead of horizontal approach to architecture composition [Bur06], it does not create new architectures and only finds a valid composition of existing architectures satisfying a high-level connector specification. All architectures must be therefore predefined and stored in a repository accessible to the generator. Consequently, to include a performance instrumentation element in connector architecture, we must define additional variants of existing architectures that will contain the performance instrumentation element, as illustrated in Figure 41. Besides modifying the architectures structurally, the declarations of non-functional properties must be updated so that an architecture featuring a performance instrumentation element will expose the declarations of NFP attributes associated with the element. This will allow the architecture resolver in a connector generator to select such architecture when a requirement for performance instrumentation is declared in a high-level connector specification.

The performance instrumentation element serves as a bridge between the measurement infrastructure and an application, providing the measurement infrastructure with design-level events from application execution. In each address space, there will be a single instance of the measurement infrastructure, which will provide a management interface to allow configuring a measurement experiment. Each instance of a performance instrumentation element will register with the infrastructure, providing it with an interface which can be used by the infrastructure to obtain metadata information and configure the behavior of the element.

In the following sections, we present the architecture of the performance instrumentation element and show how to integrate it into existing connector architectures to allow the architecture resolver to find a connector architecture satisfying NFP requirements related to performance measurement. Using this as a foundation, we describe the design of the performance instrumentation element implementation, and describe the process of the element code generation.

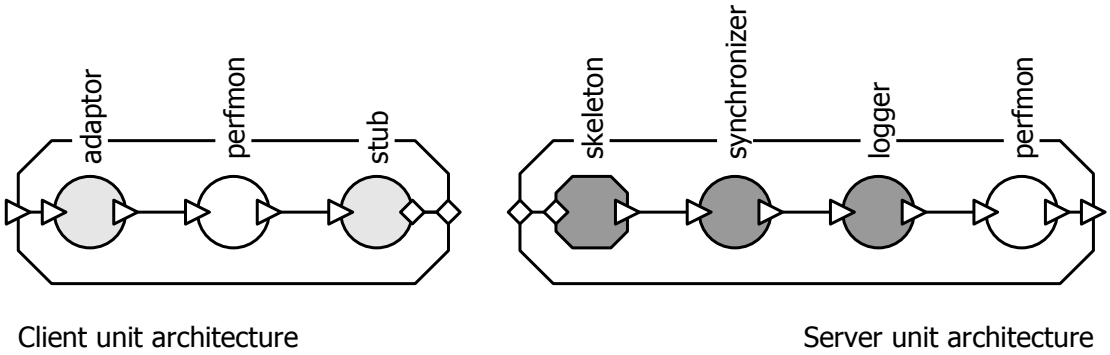


Figure 41: Connector unit architectures with a performance instrumentation element

Finally, we provide a motivation for exploiting the properties of the connector model to reduce the execution overhead of inactive performance instrumentation. This is further elaborated in Sections 5.4 and 5.5 for the general case of interception-based connector elements which provide an easy way to encapsulate optional features.

5.3.1 Performance instrumentation element in connector architectures

To enable generating connectors with performance instrumentation, the generator must have access to connector architectures featuring the performance instrumentation element. These architectures must be associated with NFP declarations which allow the architecture resolver of the connector generator to choose an appropriate architecture based on NFP requirements declared in high-level connector specification, which captures the requirements that must be satisfied by a connector realizing a particular connection among components.

Integrating performance instrumentation element into connector architectures

The basic principle of including a performance instrumentation element in connector architecture has been already illustrated in Figure 41. The figure shows architecture of a server connector unit implementing a method invocation communication style with support for distribution. An instrumentation element of type perfmon has been placed between the synchronizer element and the local required port of the connector unit.

The performance instrumentation element is basically an interceptor, which defines performance events as invocations of methods it intercepts. As shown in Figure 42, such an element defines two complementary local ports named in and out with the same interface type signature. The identical port signatures mean that the interface type of the out port is identical to the interface type of the in port. However, this is only structural information which does not say anything about semantics, e.g. that such an element actually delegates the methods calls received on the in port to an element bound to the out port. The synchronizer element shown in Figure 41 is structurally identical, but has a slightly different semantics in that it only allows a single thread to invoke methods on the entity bound to its out port. Semantics is therefore associated with an element type, which consists of a name and port signatures.

An interceptor is a type of connector element which can be used to encapsulate various features based on observation of method invocations (not only) on component interface. Consequently, element architecture may contain several interceptor elements, each implementing a different feature, as shown in Figure 43. The logger element performs

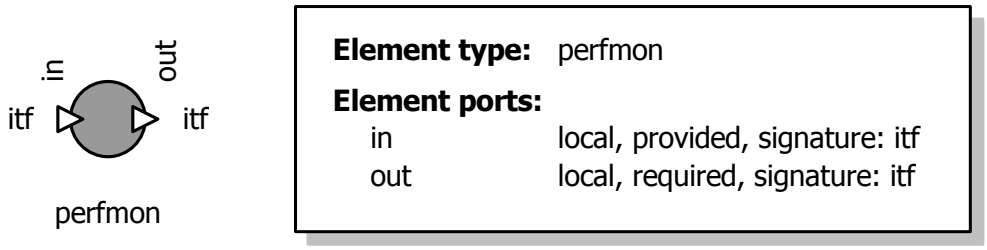


Figure 42: Performance instrumentation element with port signatures

method logs method traces, the bpchecker element performs runtime verification of behavior, and the perfmon element provides performance instrumentation. Due to the principle of vertical composition used by the architecture resolver when searching for appropriate connector architectures, providing connectors with different combinations of interceptors chained in different order in different connector units would require modifying all top-level connector unit architectures.

To avoid this situation, the enumeration of architectures reflecting different combinations and permutations of interceptor elements can be moved to a lower level of the connector architecture. This requires leaving only a single interceptor element in each of the top-level connector unit architectures, as illustrated in Figure 44, and creating a composite interceptor element containing several primitive interceptor elements. The single interceptor element in the top-level connector unit architecture may be then populated either by a primitive interceptor element, or by a composite interceptor element containing multiple interceptors if required.

Due to the properties of the architecture resolver, the various combinations and permutations of primitive interceptor elements in a composite interceptor still have to be enumerated using multiple architectures. However, this approach allows us to significantly reduce the number of top-level connector unit architectures in presence of several interceptor elements or when introducing a new interceptor element. The new element can only be reflected in the architectures of the composite interceptor, as shown in Figure 45.

A technical issue related to the connector generator requires that each primitive interceptor implementing a particular function must be defined using two element types. One of the types, interceptor, is common for all interceptors including the composite interceptor. This type is used in the top-level connector architecture, which allows using either a single primitive interceptor or the composite interceptor in the architecture interchangeably. The other type must be specific to the activity performed by a particular primitive interceptor, i.e. in case of the performance instrumentation element the type would be perfmon. These element types are intended for inclusion in the architecture of the composite interceptor.

All this is required to make the connector model finite, because if we allowed the interceptor element type to be used in the composite element, it would mean that it can be included in itself, which would lead to an infinite connector model which the architecture resolver is unable to handle. This problem could be alleviated by modifying the architecture resolver to impose a limit on the cost of an architecture, which would restrict the recursion of the model, however we prefer the finite model approach, which may need some modifications in the way connector architectures are described to the generator to avoid duplicate specification of element types.

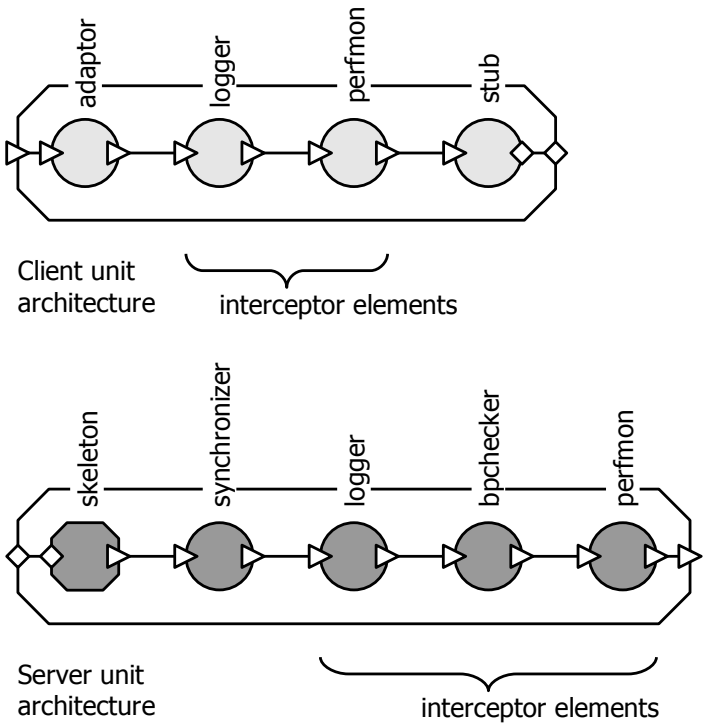


Figure 43: Connector unit architectures with multiple interception elements

Finding a suitable interceptor element architecture

Given the number of architectures of the composite interceptor, we must be able to instruct the connector generator to choose a suitable architecture for a particular purpose. The purpose needs to be expressed using NFP requirements in a high-level connector specification, and the various composite interceptor architectures must be able to express the properties of the architecture, e.g. the combination and permutation of interceptor elements, using the NFP mappings associated with each of the architectures.

To demonstrate the principle, we will use fragments of connector architecture descriptions as well as high-level connector specification. Even though it partially reveals the technical details related to the internals of the architecture resolver, it is an essential part of the connector generator that will not change in near future, in contrast to the element code generator (see below).

Specification of simple NFP attributes

The following fragment shows a high-level connector specification describing a connector consisting of two units in different deployment docks. The two units are attached to their respective components, one of them requiring a particular interface type and the other providing the same interface type. The nfp-requirement element in each of the units prescribes the communication style, in this case *method invocation*, which should be realized by particular connector unit architecture.

```
<specification>
   <unit name="server_unit" dock="dockB">
      <nfp-requirement
         predicate="nfp_mapping(Unit, 'communication_style', 'method_invocation')"/>
      <port name="call" type="required"
```

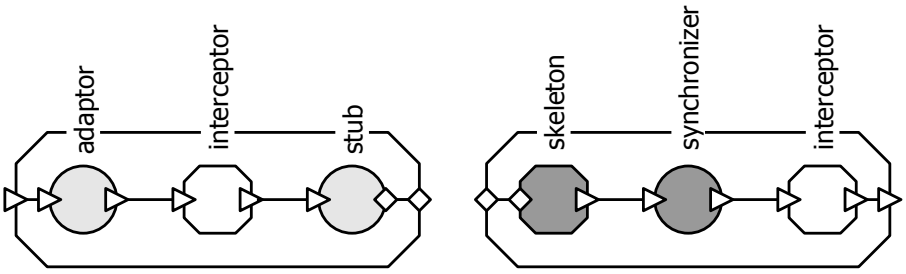


Figure 44: Connector units with a single interceptor type

```
            signature="java_interface('org.objectweb.dsrg.congen.demo.ifaces.Log')"/>
    </unit>

    <unit name="client_unit" dock="dockA">
        <nfp-requirement
            predicate="nfp_mapping(Unit, 'communication_style', 'method_invocation')"/>
        <port name="call" type="provided"
             signature="java_interface('org.objectweb.dsrg.congen.demo.ifaces.Log')"/>
    </unit>
</specification>
```

The above connector specification can be satisfied by a connector unit architecture with signature-compatible local ports that implements the required communication style. The following fragment shows an excerpt from the specification of a server connector unit without a performance instrumentation element, as depicted in Figure 41. Specifically, the purpose is to point out the contents of the nfp-declarations element, which may contain multiple nfp-mapping elements that define the values of NFP attributes. This particular architecture declares that it supports the *method invocation* communication style required in the above high-level specification.

```
<element name="server_unit" type="rpc_server_unit" impl-class="ServerUnit">
    <architecture cost="0">
        <!-- Defines instances of element types and bindings between them. -->
        ...
    </architecture>

    <nfp-declarations>
        <nfp-mapping name="communication_style" value="method_invocation"/>
    </nfp-declarations>

    <script>
        <!-- Defines actions needed to generate element implementation. -->
        ...
    </script>
</element>
```

Propagation of NFP attributes

Element architecture is only a template prescribing instances of particular element types. The slots corresponding to a particular element type may be realized by different element

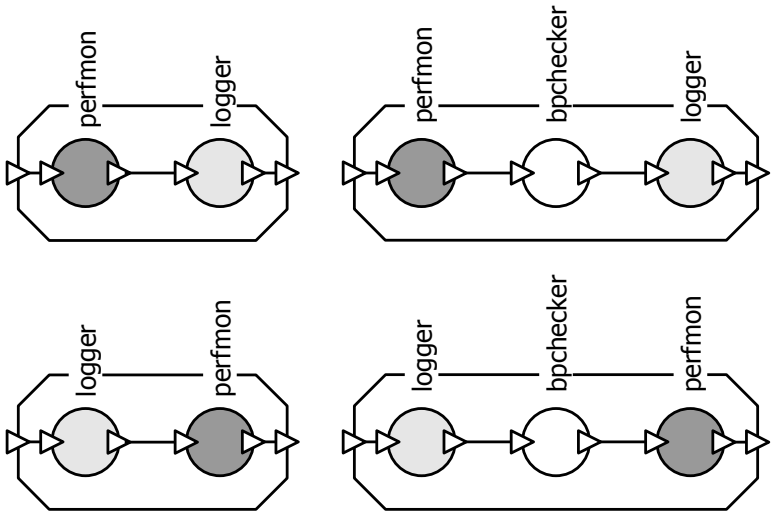


Figure 45: Variants of composite interceptor architectures

architectures, some of which can be again composite and contain additional instances of element types. In practice, particular NFP attribute is satisfied by primitive elements, which actually implement the required functionality. Consequently, NFP attributes of composite elements depend on the NFP attributes primitive elements they contain. Therefore the model must be able to propagate NFP attributes from the primitive elements (leaf nodes of architecture) to the top-level composite element, representing a connector unit.

The following fragment shows the specification of a primitive performance instrumentation element found in the server connector unit architecture depicted in Figure 41. The nfp-declarations element contains an nfp-mapping element which declares that this particular primitive element implements the perfmon functionality on a level described as simple.

```
<element name="simple_perfmon" type="perfmon" impl-class="PerformanceMonitor">
   <architecture cost="1">
        <port name="in" signature="I"/>
        <port name="out" signature="I"/>
   </architecture>

   <nfp-declarations>
        <nfp-mapping name="perfmon" value="simple"/>
   </nfp-declarations>

   <script>
        ...
   </script>
</element>
```

A connector unit containing an element implementing the perfmon functionality should also declare the perfmon NFP attribute, however with a value depending on a particular implementation of the instrumentation element. In this case, the element declares simple functionality, but there may be another implementation of the same type of element, which could declare complex or some other value of the perfmon attribute.

The propagation of an attribute value is illustrated in the following fragment, which corresponds to the specification of the server connector unit with a performance instrumentation element, again depicted in Figure 41. The nfp-mapping element for the perfmon attribute specifies a value that is derived from the value declared by the implementation of the perfmon element type. Contents of the nfp-mapping element reflect the Prolog internals of the architecture resolver. The get_elem serves to access the configuration of the perfmon element and the nfp_mapping term then serves to unify the Value atom with the value declared by the perfmon element.

```
<element name="perfmon_server_unit" type="rpc_server_unit" impl-class="ServerUnit">
   ...
   <nfp-declarations>
        <nfp-mapping name="communication_style" value="method_invocation"/>

        <nfp-mapping name="perfmon" value="Value">
             get_elem(This, 'perfmon', SE_Perfmon),
             nfp_mapping(SE_Perfmon, 'perfmon', Value)
        </nfp-mapping>
   </nfp-declarations>
   ...
</element>
```

Specifying generic interceptor NFP attributes

To define a generic interceptor as proposed earlier, the NFP attribute specifications become a bit more complex. The main idea is to allow definition of interception-based functionality in the high-level connector specification. Therefore the NFP attribute itself should be parameterized by the particular interceptor function. The following fragment shows an excerpt from the high-level specification described earlier. The specification of the client unit has been omitted, and the specification of a server unit contains additional nfp-requirement declarations, which specify that the server connector unit should support interception based logging and performance instrumentation.

```
<specification>
   <unit name="server_unit" dock="dockB">
        <nfp-requirement predicate="nfp_mapping(Unit, interceptor(logging), 'visual')"/>
        <nfp-requirement predicate="nfp_mapping(Unit, interceptor(perfmon), 'simple')"/>

        <nfp-requirement
             predicate="nfp_mapping(Unit, 'communication_style', 'method_invocation')"/>
        ...
   </unit>
   ...
</specification>
```

The specification of the server connector unit and the primitive elements implementing interception-based functions need to be updated to use a generic interceptor type and to support the parameterized NFP attribute. The architecture of a server connector unit using a generic interceptor is depicted in Figure 44.

The support for parameterized NFP attribute in element architecture is shown in the following two fragments. The first fragment contains the updated specification of a performance instrumentation element.

```
<element name="simple_perfmon_interceptor" type="perfmon" impl-class="...">
   ...
   <nfp-declarations>
        <nfp-mapping name="interceptor(perfmon)" value="simple"/>
   </nfp-declarations>
   ...
</element>
```

The second fragment contains the updated specification of the server connector unit.

```
<element name="interceptor_server_unit" type="rpc_server_unit" impl-class="ServerUnit">
   ...
   <nfp-declarations>
        <nfp-mapping name="communication_style" value="method_invocation"/>

         <nfp-mapping name="interceptor(X)" value="Value">
              get_elem(This,'interceptor',SE_Interceptor),
              nfp_mapping(SE_Interceptor,interceptor(X),Value)
        </nfp-mapping>
   </nfp-declarations>
   ...
</element>
```

However, the above high-level specification using the parameterized NFP attributes requires connector architecture to provide both logging and perfmon interception-based functions. This requirement cannot be satisfied by a server connector unit with a single primitive interceptor. As described earlier, this is addressed by defining a composite architecture for the interceptor element type which contains the required interception elements.

The following fragment shows a specification of an architecture corresponding to a composite interceptor depicted in Figure 45. The composite interceptor contains two interceptor elements, this time using element types specific to the function of each element to avoid unlimited recursion in the connector model.

```
<element name="logger_perfmon_interceptor" type="interceptor" impl-class="...">
   <architecture cost="1">
        <inst name="logger" type="logger"/>
        <inst name="perfmon" type="perfmon"/>
        ...
        <binding element1="logger" port1="out" element2="perfmon" port2="in"/>
        ...
   </architecture>

   <nfp-declarations>
        <nfp-mapping name="interceptor(logging)" value="Value">
             get_elem(This, 'logger', SE_Logger),
             nfp_mapping(SE_Logger, 'logging', Value)
```

```
        </nfp-mapping>

        <nfp-mapping name="interceptor(perfmon)" value="Value">
            get_elem(This,'perfmon', SE_PerfMon),
            nfp_mapping(SE_PerfMon, 'perfmon', Value)
        </nfp-mapping>
    </nfp-declarations>
    ...
</element>
```

Specifying the ordering of interceptor elements

The ordering of the interceptor elements in a composite interceptor is defined by the binding between the interceptor ports. In the above architecture, the out port of the logger element is bound to the in port of the perfmon interceptor; therefore the logger element is first in the chain. However, different ordering may be required for different purposes, especially when there are three or more interceptors in the composite interceptor architecture.

This can be again addressed by parametric NFP attributes, this time related to the position of a particular interception element in the interceptor chain. For illustration in the context of this work, we may want the performance instrumentation element to be either the first or the last interceptor element in the interceptor chain.

Therefore we update each of the generic interceptor architectures with NFP mappings that describe the relative ordering of the interceptor element. The following fragment shows an updated specification of the above mentioned composite interceptor, which also declares NFP attributes capturing the position of the interceptors in the interceptor chain.

```
<element name="logger_perfmon_interceptor" type="interceptor" impl-class="...">
    <architecture cost="1">
        <inst name="logger" type="logger"/>
        <inst name="perfmon" type="perfmon"/>
        ...
        <binding element1="logger" port1="out" element2="perfmon" port2="in"/>
        ...
    </architecture>

    <nfp-declarations>
        ...
        <nfp-mapping name="interceptor-position(logging)" value="first"/>
        <nfp-mapping name="interceptor-position(perfmon)" value="last"/>
    </nfp-declarations>
    ...
</element>
```

Primitive architectures of the generic interceptor element type that are expected to be interchangeable with the composite interceptor will declare that the interceptor occupies both first and last positions.

The high-level connector specification will then use the interceptor-position NFP attribute in addition to the interceptor attribute to specify the desired interception-based functions and the ordering of important interceptors in the call path. This is shown in the following,

and last, fragment of high-level connector specification, which requires the performance instrumentation element to be the last in the chain, therefore the closest to the component interface.

```
<specification>
   <unit name="server_unit" dock="dockB">
        <nfp-requirement predicate="nfp_mapping(Unit, interceptor(logging), 'visual')"/>
        <nfp-requirement predicate="nfp_mapping(Unit, interceptor(perfmon), 'simple')"/>

        <nfp-requirement
             predicate="nfp_mapping(Unit, interceptor-position(perfmon), 'last')"/>
        ...
   </unit>
   ...
</specification>
```

The above approach shows how to employ the NFP attribute requirements and mappings supported by the architecture resolver to express requirements on interception-based functions in a high-level connector specification. Besides a particular function, the specification allows expressing requirements for a particular ordering of important interception elements, such as the performance instrumentation element. Admittedly, the mapping of NFP attributes is still rather low-level and depends is specific to the implementation of the architecture resolver. However, the basic concept is sound, only the architecture resolver should support capturing the same concepts using a less implementation-specific description.

5.3.2 Design of the performance instrumentation element

As presented in the previous section, the semantics of the performance instrumentation element corresponds to an interceptor. As such, the performance element type defines two local ports, a provided in port and a required out port, with identical signature. This allows including the element anywhere in connector architecture, however for the purpose of this thesis, we aim to include it in the call path of component interface methods.

The diagram in Figure 46 shows the context of the runtime environment in which a performance instrumentation element will operate. The left part of the diagram captures the integration of the instrumentation element (represented by the PerformanceElement class) with the connector runtime environment. Since it defines two local ports, it has to support the ElementLocalServer and ElementLocalClient interfaces which allow the connector runtime to query and bind port references.

Since the performance instrumentation element only defines local ports, it cannot appear in connector architecture in place of an element representing a connector unit. Consequently, there will be always a parent element responsible its instantiation and initialization; in most cases this will be an element representing a connector unit. This is reflected in the diagram using a ConnectorUnit entity responsible for creating an instance of the performance instrumentation element.

The right part of the diagram in Figure 46 captures the integration of the performance instrumentation element with the measurement infrastructure (represented by the MeasurementInfrastructure, see Figure 31 on page 87 for reference) which has to reside in the same address space as the instrumentation element. The measurement

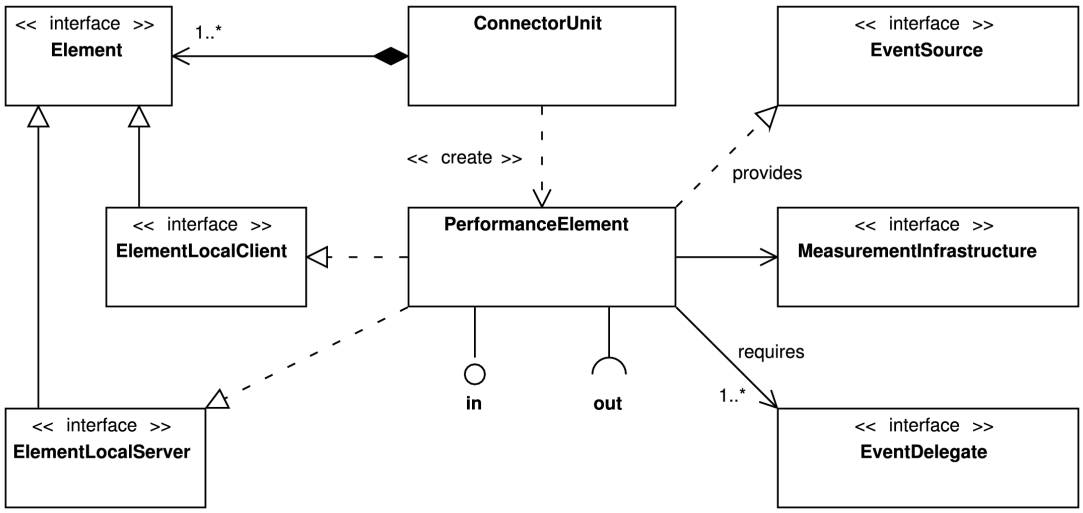


Figure 46: Runtime context of a performance instrumentation element

infrastructure uses the EventSource interface to communicate with the instrumentation element, and in turn the element uses the EventDelegate interface to provide the measurement infrastructure with performance events.

As with other connector elements, the performance instrumentation element is expected to be generated. Therefore the main goal in designing such an element is to keep it simple so that its code generator can be kept simple as well. This in turn simplifies the process of extending the generator to support generating the element for more target programming languages.

The measurement infrastructure described in Chapter 4 was designed to allow separating the functionality required for performance measurement and for performance instrumentation. Therefore the performance instrumentation element will be only responsible for the following tasks necessary to provide the measurement infrastructure with performance events:

1. initializing the measurement infrastructure, provided it has not been already initialized by other element instance, and register itself with the measurement infrastructure,
2. providing a management interface to the measurement infrastructure so that it can
   - uniquely identify the location of performance events in application architecture and associate it with a particular component instance and component interface,
   - query the performance events supported by an element,
   - provide an element with event handlers to be used fore reporting performance events, and
   - configure which events should be reported by an element to the measurement infrastructure, and finally
3. reporting configured performance events during application execution using the event handlers provided for specific event types.

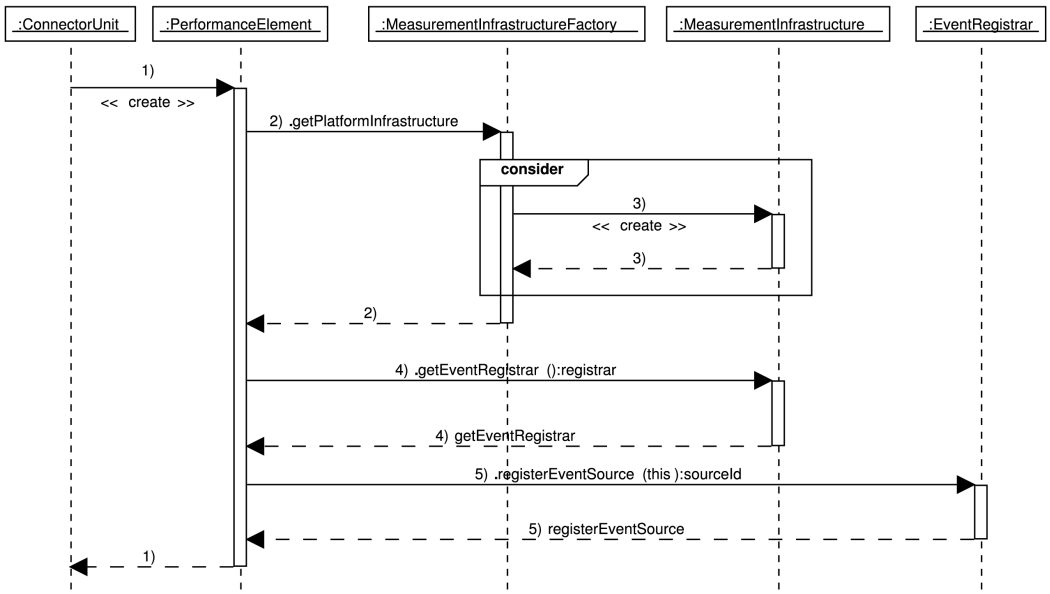


Figure 47: Performance instrumentation element initialization

Conceptually, these tasks correspond to different operational stages of the instrumentation element, which will refer to as element initialization, element configuration, and element operation.

Element initialization

Element initialization is the first operational stage of a performance instrumentation element after it has been instantiated by its parent element. In this stage, after initializing its private data structures, the element registers with the performance infrastructure and provides it with a management interface for further communication.

This process is illustrated in Figure 47. A connector unit (or another composite element in general) creates an instance of the performance instrumentation element. Since the connector model does not separate element instantiation from its initialization, the initialization is performed in the constructor of a generated class providing the implementation of the element adapted to a particular component interface.

During initialization, the element uses a MesurementInfrastructureFactory class to obtain an instance of the MeasurementInfrastructure interface. The factory provides as helper method to ensure that there is only a single instance of the measurement infrastructure in each address space. When first called, the factory creates a new instance of the infrastructure and stores the reference for subsequent calls.

After obtaining a reference to the measurement infrastructure, the performance instrumentation element obtains a reference to the EventRegistrar interface and registers with the infrastructure under a name reflecting its position in application architecture and provides the infrastructure with an EventSource interface implemented by it. The interface will used by the measurement infrastructure for subsequent communication. Upon registering the instrumentation element as an event source, the measurement infrastructure assigns the element a unique event source identifier which it will use when reporting events to the infrastructure.

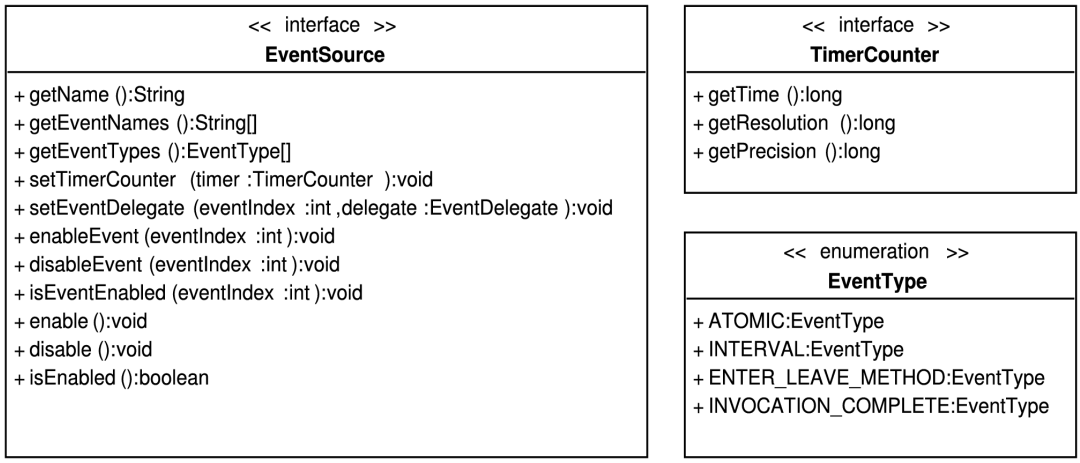


Figure 48: Detail of EventSource interface and related entities

There is a minor issue with the design of the connector environment runtime, since it does not provide a method for explicit element initialization after its construction. If an implementation of the element runtime performs element initialization during its construction, there is an inherent and hard to avoid race condition. An element would typically provide a reference to itself when registering with the infrastructure; however doing so from constructor an element may provide the infrastructure with a reference to an object that has not yet been constructed. This problem needs to be resolved in the connector runtime environment by decoupling the element instantiation from element initialization. Concerning the model of connector runtime, the Element interface should be extended to provide an explicit element initialization method.

Element configuration

After initialization and registration with the measurement infrastructure, a performance element has to provide the infrastructure with information necessary to identify the location of the element in the application. This process is initiated by the measurement infrastructure, using the EventSource interface provided by an element during registration.

Figure 48 shows the EventSource interface, along with an EventType class and a TimerCounter interface. The measurement infrastructure uses getEventNames and getEventTypes methods to obtain the names and types of all events related to method invocations supported by the element as well as any other events that may be related to the element instance.

The TimerCounter interface is intended to provide the instrumentation element with a precise time source, because for some types of performance events, it may be beneficial to collect the timestamps within an interceptor. Providing a time source to an element is also important to ensure that all timing information collected within the measurement infrastructure has been obtained from a common source with known resolution and precision. Without an external time source, the instrumentation element may decide to use a native time source or not collect time stamps at all.

The diagram in Figure 49 captures the element configuration process driven by the measurement infrastructure. First, the measurement infrastructure calls the getName method to obtain the event source name. Then it determines the names and types of

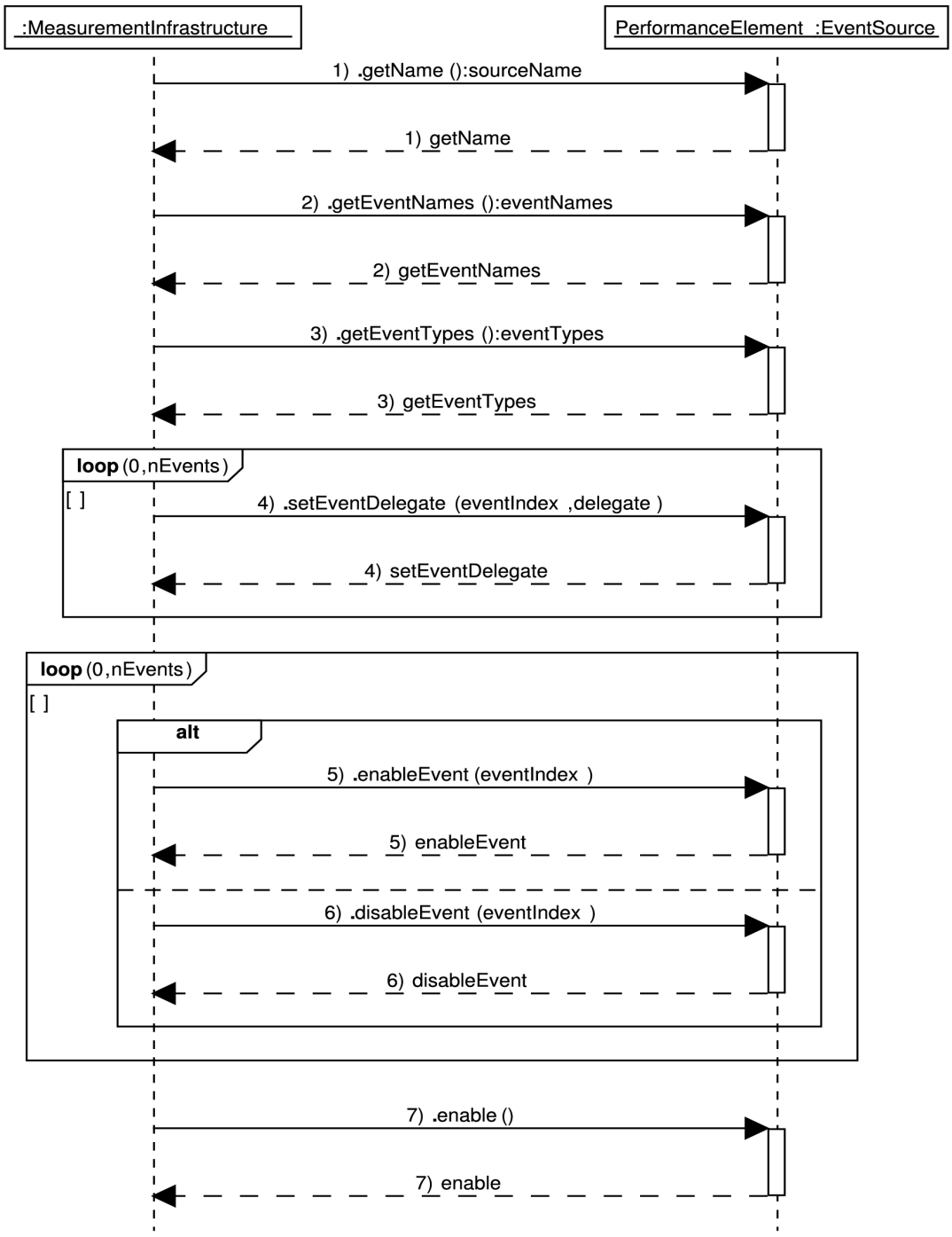


Figure 49: Configuration of a performance instrumentation element

performance events supported by the instrumentation element by calling the getEvent-Names and getEventTypes methods and examining the returned arrays. The ordering of event names array is important, because it identifies an event in the context of a particular event source. The event index will be used in subsequent calls related to individual events, specifically to configure which events should an element report to the infrastructure.

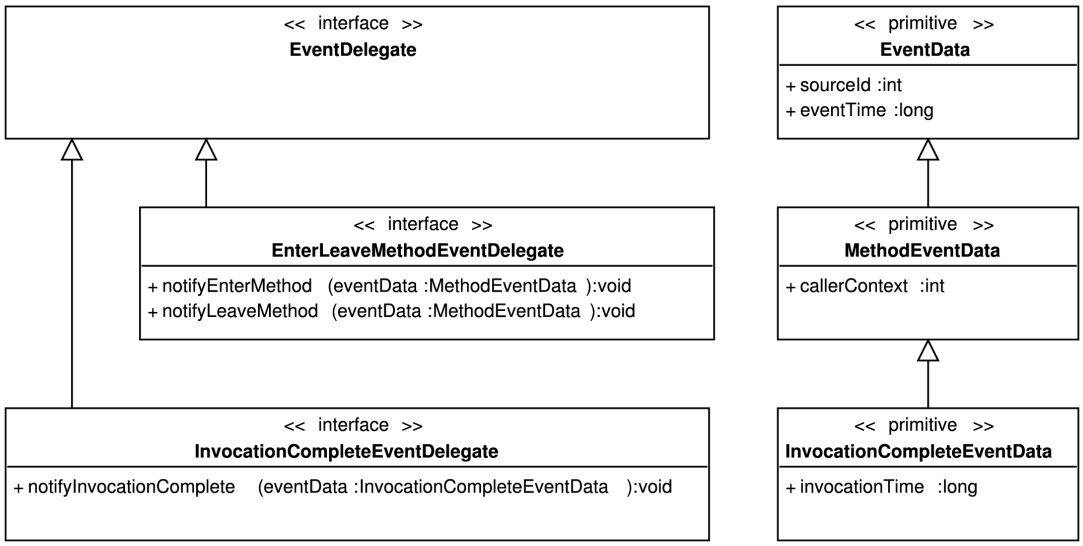


Figure 50: Structure of event delegates

The event source name and the names of supported events may be used by the infrastructure to construct a management model of an application, which in turn can be used by management tools to configure performance measurement experiment.

For each supported event, an element needs to be provided with an event delegate, which will process performance event notifications emitted by the element. Since each supported event can be of different type, the types of the provided event delegates must correspond to the respective event types. The structure of event delegates is shown in Figure 50. Each event delegate is intended to handle one or more semantically related event notifications. The figure shows two selected types of event delegates. The EnterLeaveMethodEventDelegate supports event notifications corresponding to entering and leaving a method invocation, while the InvocationCompleteEventDelegate only supports a single notification corresponding to a completed method invocation. Each event delegate may associate different data with different events. Even though in practice both event delegates provide equivalent timing information, the separate notifications of the EnterLeaveMethodEventDelegate interface let the event delegate perform different actions with each notification.

The event delegates are expected to correspond to runtime entities responsible for processing and storing event data. Even though the events could be defined in a generic way, the approach with specific event delegate interfaces was chosen to avoid dispatching events according to their type within the measurement infrastructure.

When the infrastructure obtains the necessary information about a particular instrumentation element, it may configure (typically in response to external requests) the element to emit selected performance events. To enable particular event notifications, the infrastructure has to provide an element with event delegates for respective events using the setEventDelegate method and then enable emitting particular event notifications by calling the enableEvent method on the element's EventSource interface. This is captured in the middle section of the diagram in Figure 49. Finally, to start receiving any performance events, the measurement infrastructure has to enable the interception functionality of the instrumentation element by calling the enable method.

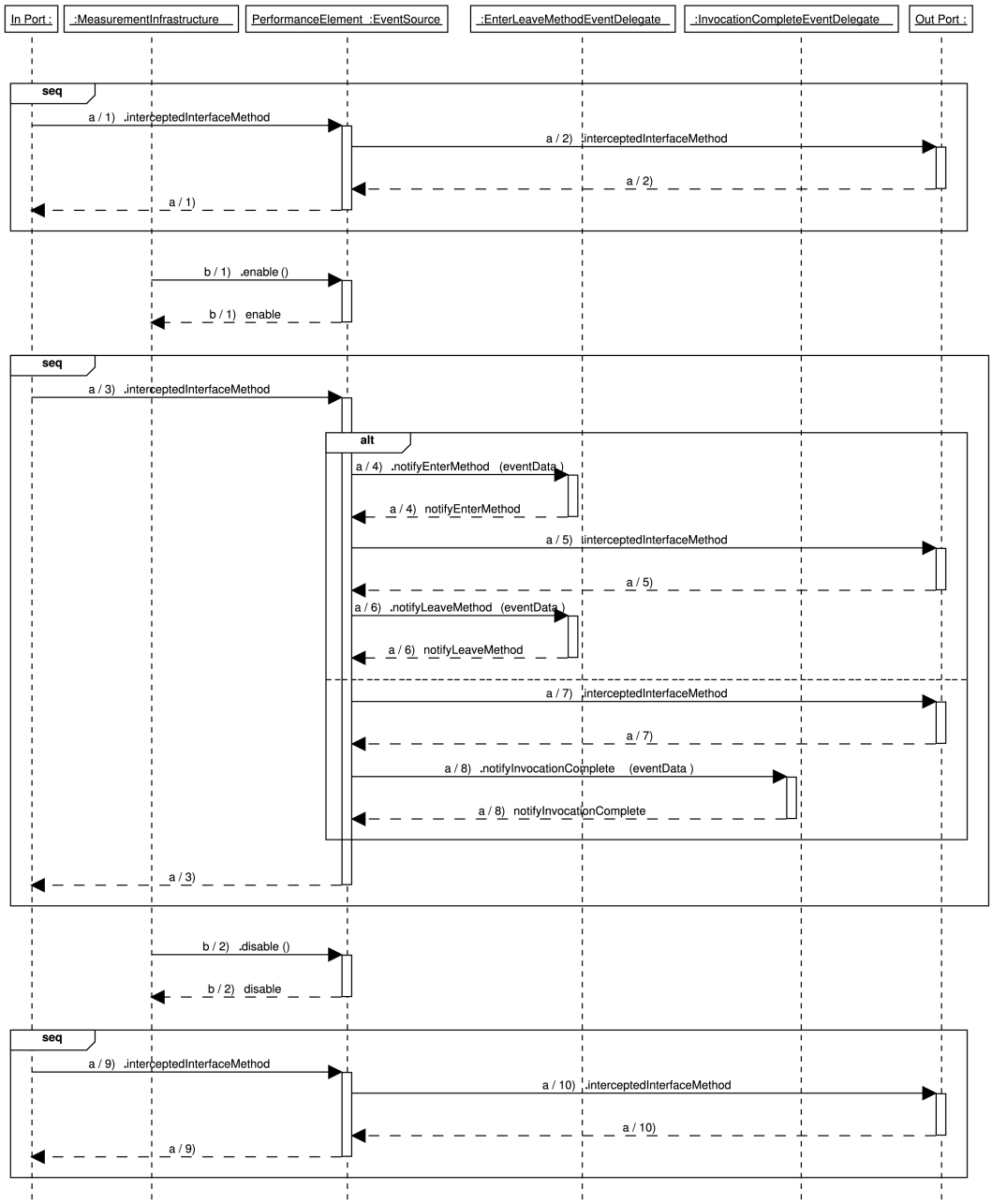


Figure 51: Performance instrumentation element operation

Element operation

Element operation is the last conceptual stage in execution of a performance instrumentation element. In this stage, the element should have been configured by the measurement infrastructure to provide it with selected performance events for selected component interface methods. The element intercepts method invocations on its provided in port and forwards them to a target object bound to its required out port. Depending on the configuration, performance event notifications corresponding to method invocations are reported to the measurement infrastructure.

The process is illustrated in Figure 51. The interception functionality of the element is initially disabled; the element therefore passively forwards component interface method

invocations from its in port to its out port. This is captured in the top part of the diagram as a sequence of method invocations initiated by an entity bound to the in port of the element.

When the management infrastructure is instructed to collect performance data at specific locations, it enables the interception functionality of performance instrumentation element in these locations. This provides the infrastructure with events required to obtain the desired performance data. The configuration of measurement infrastructure is initiated from outside, and is not explicitly shown in the figure. However, this results in the MeasurementInfrastructure entity invoking the enable method on the respective instance of a PerformanceElement which enables the interceptor functionality of the element.

Subsequent method invocations on the in port of the element are then intercepted before being forwarded to the out port. Depending on the configuration, the element either issues event notifications before forwarding the invocation to the target and after it has returned, or it issues only single event notification after the invocation has returned. This is reflected in the middle part of the diagram by a sequence of method invocations initiated by an entity bound to the in port of the element, but in this case there are two alternative ways of processing, depending on the configuration. This mode of operation lasts until the measurement infrastructure disables the interception functionality of the element.

Instead of enabling and disabling the element interception functionality using the enable and disable methods, the measurement infrastructure may be instructed to enable or disable event notifications only for specific methods. This can be done using the enableEvent and disableEvent methods to configure which event notifications should be reported to the infrastructure. If no events are enabled, none are reported. Even though we have described the element configuration and element operation stages separately, the invocations on the EventSource interface may occur any time during element operation.

5.3.3 Generating performance instrumentation element code

When the architecture resolver part of the connector generator determines the connector configuration that should be used for a particular connector, the element generator part is responsible for generating the implementation code of connector elements, including the performance instrumentation element.

The code of the performance instrumentation element has to reflect the design presented in the previous section. Since writing code to write other code is generally difficult, the goal of the design was to move maximum functionality related to collection of performance data to the measurement infrastructure, leaving only minimum responsibilities on the instrumentation layer. This leaves the performance instrumentation element simple which significantly eases the writing of its code generator.

The implementation of the instrumentation element does not depend on a particular component system, because the connector runtime environment shields the connector elements from such influence. This does not include type systems specifics, these are however handled by type factories in the generator and the elements are designed to be adaptable in this respect. The code generator is therefore responsible only generating element code for a particular programming language. Even though separate generators are required for different programming languages, creating a new code generator should not pose any significant problem due to simplicity of the instrumentation element code.

The code generator originally [Bur06] consisted of an element code template for a particular programming language and a plug-in class for a generic code generator, which was responsible for expanding the template. The code template provided the static part of element code, which was not affected by element adaptation, while the expander class provided the dynamic part of the code, reflecting the adaptation undergone by the element. This mechanism was later replaced by a more powerful template language with expansion engine based on Stratego/XT [DVV01, STRXT]. The new template mechanism supports including template fragments common to multiple connector elements and allows using generative constructs directly from the template.

The code generation part of the generator is still evolving, therefore describing a particular element code template and its expansion process is not very future proof. However, regardless of the code generation method, most of the instrumentation element code can be generated statically, dynamic generation is only required to reflect the element adaptation, to generate the implementation of intercepted methods providing performance events, to configure data structures required for management purposes, and to provide the element instance with metadata needed to construct the event source name as well as the names of method related events for unique identification of event locations in application architecture.

5.3.4 Managing instrumentation overhead in connectors

In the context of the architecture-based connector model, the performance instrumentation code represents an optional functionality that can be activated and deactivated at runtime. This is important for applications running in a production environment, because collecting performance data may impose a signification overhead on application execution. Unless there is a problem that needs to be diagnosed, the performance instrumentation in an application should be either disabled, or enabled only at selected locations for steady-state execution monitoring.

However, even inactive optional code is a source of certain overhead, which is associated with the mechanism that allows activating and deactivating the code at runtime. Such mechanisms are typically implemented directly using conditional statements to control the flow of execution, or indirectly using a chain of indirect method invocations. A mechanism based on conditional execution is less flexible, because the optional code is restricted to a particular location in the application. On the other hand, a mechanism based on indirect invocation is more flexible, because it allows composition of optional code, but its overhead increases with every layer of indirection. This approach is used for chaining connector elements in the call path of component interface methods, and consequently for including performance instrumentation code in connectors. Since there can be potentially several intercepting elements in connector unit architecture, this may result in extraneous indirections in each method invocation should the intercepting elements be passive.

To encourage performance instrumentation of production applications, we aim to eliminate the overhead associated with inactive instrumentation code, so that an instrumented application can run completely unhindered by performance data collection as long as the instrumentation is disabled. To achieve that, we have extended the connector runtime environment to provide a reconfiguration mechanism that allows complete exclusion of the performance instrumentation code from execution. This can be done automatically when a connector element responsible for performance instrumentation detects that is has been configured to emit no performance events.

5.4 Performance Instrumentation Overhead

Any instrumentation code injected into an application usually incurs certain overhead, both in terms of execution time and memory consumption. For both types of overhead we distinguish between their static and dynamic part. Static overhead is associated with the instrumentation mechanism used to inject instrumentation code into an application, which ensures that the instrumentation code is included in application execution. Dynamic overhead is associated with the activity performed by the instrumentation code. Since the real functionality is contained in the instrumentation code, the dynamic overhead is the major source of the overall instrumentation overhead.

In some situations, the instrumentation code is intended to be always included in application execution, e.g. in debugging or profiling during application development. In such cases, there is no need to distinguish between static and dynamic overhead when assessing the overhead imposed by the instrumentation, because the two cannot be separated. However in other situations, the instrumentation is expected to be activated and deactivated at runtime with the instrumentation disabled at launch, such as e.g. performance instrumentation or trace logging. In such cases, the distinction between static and dynamic overhead is necessary, because when assessing the overhead of the instrumentation, we may be interested in the overhead caused by the instrumentation even when it is disabled. In principle, static overhead represents the cost of presence of the instrumentation in an application.

Distinguishing between static and dynamic overhead is also important because the two kinds of overhead can be managed separately. Dynamic overhead is caused by the instrumentation code implementing a particular function, and we attempt to limit its impact on the running application. Static overhead is caused by the instrumentation mechanism, and we attempt to minimize it or, ideally, eliminate it when the instrumentation code is inactive.

5.4.1 Overhead of enabled performance instrumentation

When the performance instrumentation code is enabled, it collects performance information whenever there is activity on the instrumented component interfaces. Naturally, this causes certain overhead compared to the execution of a non-instrumented application, but since we cannot hope to observe a system without influencing it, we aim to keep the overhead at minimum or within reasonable bounds, depending on the effort we can afford to spend on the task.

Memory overhead

Static memory overhead is associated mainly with the additional code the instrumentation adds to an application. The instrumentation code consists of the measurement infrastructure and an implementation of a performance instrumentation connector element. The infrastructure collects performance data and provides a management interface that allows configuring performance experiments at runtime. Consequently, it provides an interface to enabling or disabling either parts or the entire instrumentation. The instrumentation element provides the infrastructure with performance events and allows the infrastructure to select which events it is interested in. If no events are reported, the instrumentation becomes passive. Dynamic memory overhead is caused mainly by the measurement infrastructure, which needs to store the performance data it collects before delivering it to consumers.

Since the performance instrumentation elements are designed to have a simple implementation, most of the code implementing the performance measurement is contained in the measurement infrastructure. Even though the measurement infrastructure is more complex than a performance instrumentation element, we do not expect the overall amount of instrumentation code exceed more then a small fraction of all application code, except for tiny applications. The instrumentation code may still be an issue for embedded systems, but these constitute a special case for which, should the code size really become an issue, a simplified version of the measurement infrastructure could be provided.

Dynamic memory overhead caused by the measurement infrastructure plays a more vital role in the overall memory overhead. Since the measurement core needs memory to store the collected data, the required amount of memory grows with the number of observed performance data values associated with a performance event, the number of observed performance events, and finally the number of event records that need to be kept in memory. These numbers are configurable; therefore the dynamic memory overhead can be managed from outside the performance measurement framework and thus becomes the responsibility of the framework user. The measurement infrastructure should only avoid wasting memory.

Execution overhead

Static execution overhead is associated with the inclusion of an additional element in connector architecture (see Figure 41). The architecture determines the bindings between connector elements and therefore the call path for method invocations mediated by a connector. Since there can be multiple interception-based elements similar to the performance instrumentation element (see Figure 43), the invocation of a component interface method may need to pass through several interceptor elements.

Since the performance instrumentation element only provides performance event, the code intercepting component interface methods is very simple. Before forwarding a method invoked on its input port to the target bound to the output port, the element checks if the measurement infrastructure is to be notified about any events associated with the start of method invocation and emits the necessary events if necessary. Then it proceeds with forwarding the method invocation to its output port. After the invocation completes the element checks again and emits necessary performance events if there are any. These checks and an additional level of indirection in method invocation represent the static execution overhead associated with a performance instrumentation element.

Dynamic execution overhead is associated with the measurement infrastructure, which has to collect performance data and store them in memory. Upon exhausting the storage capacity, the measurement infrastructure proceeds according to a configured policy to obtain free storage capacity for subsequent data.

When notified about a particular performance event, the measurement infrastructure collects values of performance data associated with the event, and optionally performs compression, aggregation or other transformations to reduce the amount of data that needs to be stored and later delivered to consumers. These operations are more complex than taking timestamps and delegating method invocations done by a performance instrumentation element, therefore the measurement infrastructure is the main source of execution overhead associated with performance instrumentation.

The amount of memory used for storing the collected performance data needs to be balanced with respect to the execution overhead. Allocating too much memory for storing

performance data may result in excessive processor cache trashing or in case of virtual machine environments, excessive garbage collector activity, which may incur additional execution overhead or cause e.g. large bursts of data being sent to the network. On the other hand, allocating too little memory may result in too frequent flushing of data buffers, which may again incur additional execution overhead and e.g. cause considerable amount of packets to be sent to the network.

Reducing the amount of data by compression or filtration may help keep the memory requirements and the amount of data sent over a network in reasonable bounds, but it may also incur additional execution overhead and exhibit an unwanted influence on the system [Can06]. However, if the complexity of the data reduction operations is comparable to that of data storage, the execution overhead may be actually beneficial because the data will need to be stored in less memory, reducing the chance of trashing processor cache. This balancing, however, is a matter of policy and as such should be left in the hands of a user of the measurement infrastructure.

5.4.2 Overhead of disabled performance instrumentation

Ideally, there should be no overhead during execution of an instrumented application when the instrumentation code is disabled. It should not consume any memory and cause no delays in execution. However, in general this is difficult to achieve, especially in case of static overhead, which is always present. Eliminating it would require removing the instrumentation from an application. Since static overhead is typically less significant than dynamic overhead, it is often tolerated in exchange for the benefits the presence of instrumentation code provides in production environment.

Leaving instrumentation code in production applications is important to allow on-site diagnosis of problems that are hard to reproduce in a development environment. This is especially true for distributed applications.

Memory overhead

We consider the performance instrumentation disabled when no performance instrumentation element is configured to report performance events. In such state, the measurement infrastructure part of performance instrumentation is no longer processing events, but remains in memory and services the management interface exposed to the outside. Disabling performance instrumentation therefore does not change static memory overhead associated with instrumentation code. However, since the overall amount of instrumentation code is expected to be only a small fraction of application code, the static memory overhead should be negligible.

The dynamic memory overhead is caused primarily by the memory allocated for storing performance data. When the performance instrumentation is disabled, the measurement infrastructure can free this memory (in Java environment, all references to the memory are invalidated so that a garbage collector can free it). Some memory is needed to preserve the configuration of the measurement infrastructure and is not therefore released. However, this unreleased memory is hardly significant when compared to storage capacity needed for performance data; therefore the dynamic memory overhead should be also negligible.

For systems with constrained memory resources, additional effort can be dedicated to further reducing the memory overhead of the entire performance measurement framework.

Execution overhead

With performance instrumentation enabled, the overall execution overhead is dominated by the dynamic overhead of the measurement infrastructure. The code in the performance instrumentation element is intended to be generated so that for each component interface method, associated performance events are reported to the measurement infrastructure conditionally, only when requested. When all performance instrumentation elements are configured to report no performance events, the measurement infrastructure is not driven by the performance events, which eliminates the main cause of execution overhead.

With the performance instrumentation disabled, all the conditions evaluate to false, but still need to be always evaluated before forwarding a method invocation to the target bound to the element output port. This happens in the call path of every component interface method. These unnecessary condition evaluations along with an additional layer of indirection in component interface method invocation represent the static execution overhead associated with performance instrumentation.

Considering a connector with only a single element behaving this way, this may be considered a reasonable trade-off for the benefits of performance instrumentation and the overhead may be tolerated. However, connector architecture consists of multiple elements and some of them may also provide interception-based functionality, similar to the performance instrumentation element (see Figure 43). The guard code in all interception-based elements will always evaluate the conditions enabling their function, even when the function has been disabled. Each element will also add a level of indirection to invocation of component interface methods. Combined, all inactive elements in the call path of component interface methods may incur a measurable execution overhead.

We show that the explicit architecture of connectors and its runtime representation allow us to modify a connector instance at runtime to completely eliminate the static execution overhead caused by inactive interception-based elements in the call path of component interface method invocations.

5.5 Eliminating Overhead of Disabled Instrumentation

Since the dynamic execution overhead as well as most of the dynamic memory overhead disappears when the performance instrumentation is disabled, we focus mainly on reducing or eliminating static overhead caused by integration of the performance instrumentation with an application.

Memory overhead

The dynamic memory overhead remaining after releasing memory for storing performance data is primarily associated with instance variables of the measurement infrastructure holding the configuration of a measurement experiment and the structure of the instrumented application. This configuration should be preserved to simplify and speed up setup and activation of subsequent performance measurement experiments. However, if necessary, the configuration can be externalized or discarded, which would allow freeing the measurement infrastructure instance memory. In a virtual machine environment with garbage collection, we would have to explicitly discard references to the measurement core instance so that the memory can be freed by a garbage collector.

Reducing static memory overhead associated with performance instrumentation code requires evicting as much code as possible from memory, which is generally difficult and may be feasible only in virtual machine environments with garbage collection, which by

design do not provide any explicit control over memory management. Code eviction in such environments typically concerns only long unused just-in-time compiled code, as opposed to e.g. byte code of a Java class. Even if unloading of byte code was supported, it would require tremendous effort to persuade a garbage collector to unload class code, an effort which is not in the least justified by the results.

Moreover, static memory overhead is typically negligible when compared to its dynamic counterpart, and since it does not directly affect the execution of an application, its influence on application execution should be negligible, if at all measurable.

Execution overhead

In contrast to static memory overhead, static execution overhead is caused by code actually executing on behalf of an instrumented application and therefore has more potential for influencing its execution.

Performance instrumentation that can be enabled and disabled at runtime represents an optional feature. Generally, optional features must be supported by the surrounding code and typically there are predefined locations where optional features are supported. At these locations, the normal (non-optional) code evaluates a condition to decide whether to execute the optional code or not, or invokes callbacks using references from a collection to which the optional code registered using an application specific API. Both methods are inflexible because the optional code can be only placed at predefined places, and even if there is no optional code to execute, the surrounding code has to evaluate one or more conditions to find out.

Optional code can also be integrated with other code through a layer of indirection in method invocation, which is especially suitable for data transformations and interception-based features. To intercept a group of methods defined in an interface, an interceptor needs to expose a facade that is type compatible with the interface type and provide its own implementation of those methods.

Besides providing the interception-based functionality, the implementation delegates method invocations either to the real implementation or to another interceptor in chain. An interceptor typically remains in the call path even when the feature it provides needs to be disabled, because excluding an interceptor from a call path requires modifying references in other objects which the interceptor cannot easily access. Therefore the implementation of an interceptor-based feature executes the code of the feature conditionally.

As mentioned earlier in connection with connector elements, if there are several interceptors in a chain, each interceptor adds a level of indirection to method invocation as well as unnecessary evaluation of conditions enabling the execution of code implementing a particular feature.

The architecture-based connector model which we use for performance instrumentation of component-based applications maintains a runtime representation of its design architecture and provides programmatic access to its structure at runtime. This can be used to manipulate element port references at runtime and consequently achieve exclusion or inclusion of an interceptor element from the call path of component interface methods. This in turn allows us to completely exclude performance infrastructure from application execution.

5.5.1 Reconfiguration mechanism

Eliminating static execution overhead of an interception-based feature associated with a particular component interface requires removal of an element implementing the feature from a chain of elements intercepting method invocations on that interface.

This operation is similar to removal of an item from a single-linked list – the predecessor of the item must be provided with a link to a new successor. However, in case of connector architecture, the items in the list are not data but code. The problem thus gets complicated by the fact that connectors are distributed entities and that each of the elements in a connector is autonomous in deciding (anytime during execution) when it wants to be removed from the call chain and when it wants to be included again. Additionally, the connector architecture is hierarchical, which further complicates the management of the call chain.

Connector elements in a chain intercepting a particular component interface are linked to each other using a pair of complementary ports. Their required ports are bound to ports of the same type provided by their successors. Since each connector element is aware of its internal architecture but not of its place in the architecture of its parent element, the binding between ports of two sibling elements has to be established from outside, by a parent element responsible for that architecture level. Based on its architecture, the parent element queries its child elements for references to their provided ports and distributes these references to other child elements that require them.

Thus, when an element wants to be excluded from a call chain, it can simply realize it by providing the target reference associated with its required port as a reference to its own provided port. In other words, when queried for a reference to its provided port, an element returns the reference it has already obtained as a target for some of its required ports. Method invocations on the element's provided port are thus by-passed and invoked directly on the provided port of the next element in the call chain.

Although the trick for excluding elements from the call path is simple, there are several problems that complicate its usage

a) for initial configuration of a connector architecture at startup, and

b) for reconfiguration at runtime, because of the above mentioned autonomy in deciding whether an element wants to be included in the interceptor chain.

When creating an initial connector architecture (with some of the optional features disabled by default), the elements need to be bound in particular order for the idea to work. The order corresponds to a breadth-first traversal of a graph of dependencies among the ports. This has to be done recursively for all levels of the connector architecture hierarchy, because through delegation and subsumption between the parent and child elements, the graph of dependencies between the ports may cross multiple levels of the connector architecture hierarchy. Additionally, the concept of connector elements is based on strict encapsulation, therefore the child elements appear to their parent entity as a black-box. Consequently, there is neither central information about the entire connector architecture (viewed as a white-box), nor is it possible to explicitly construct the dependency graph for the whole connector.

In our approach, we address both situations by a reconfiguration process initiated by within a connector element. The reconfiguration process is initiated when an (originating) element determines that, for some reason, a reference it has exported to the environment as an implementation of its provided port is no longer valid and a new reference should be

used instead. To achieve this, the element would have to provide the new reference to all neighboring elements bound to its provided port using the old reference (through their required ports). However, the originating element does not have information about its neighbors. This information is available in only in the parent element responsible for the composition of its architecture.

The originating element therefore notifies its parent entity (the containing element or the connector runtime) that a provided port reference is no longer valid. The parent entity then uses its architecture information to find the neighboring elements that are bound to the provided port of the originating element, and are thus affected by the change of the reference. If there is a delegation leading to the originating element, the parent entity must propagate the notification to the higher levels of connector architecture by notifying its own parent entity.

This reconfiguration process may trigger other reconfigurations when a depending element is excluded from the call chain and thus the change of the target reference on its required port causes a change of a reference to its provided port.

The reconfiguration process provides the following solutions to the above two situations. In case of (a), we do not impose any explicit order on instantiation and binding of connector elements. When an element that is supposed to be excluded from the call chain is asked to provide a reference which it does not have (because its required port has not yet been bound), it returns a special reference UnknownTargetReference. As soon as the required port of that element is bound and the target reference (that should have been returned for the provided port) becomes known, the element initiates the reconfiguration process for the affected provided port, which ensures propagation of the reference to elements bound to that port.

In case of (b), the inclusion/exclusion of an element in/from a call chain affects a reference provided by a particular provided port. An element that is to be included in a call chain has to provide a reference to an internal object (or itself) implementing the provided port interface. This allows the element to intercept method invocations to provide a particular interception-based function. An element that is to be excluded from a call chain has to provide a reference associated with a target object of a complementary required port. In both cases, the reconfiguration process is used to propagate the change in provided port references to the elements bound to a particular provided port.

The original connector runtime environment as presented in Section 5.2.2 does not support the mechanism presented in this section. Specifically, the entities in the model cannot notify their parent entities about changes in port references. Figure 52 shows two additional interfaces we have added to the runtime model of connector environment. The ReconfigurableElement interface serves to distinguish normal connector elements from elements which support the reconfiguration mechanism. The interface contains a single method, setElReconfigurationHandler, which can be used to provide an element with a link to an entity that will handle its reconfiguration requests. The ReconfigurationHandler interface, on the other hand, has to be implemented by entities capable of handling reconfiguration requests. The interface contains two methods, invalidateElPort and invalidateElRemotePort, which can be used by a reconfigurable element to notify its reconfiguration handler about changes on its local provided and remote ports.

Reconfigurable primitive elements in connector architecture will only implement the ReconfigurableElement interface, because it does not have any child element which would need a reconfiguration handler. Reconfigurable composite elements will typically implement both interfaces, because they will have to handle requests from its child

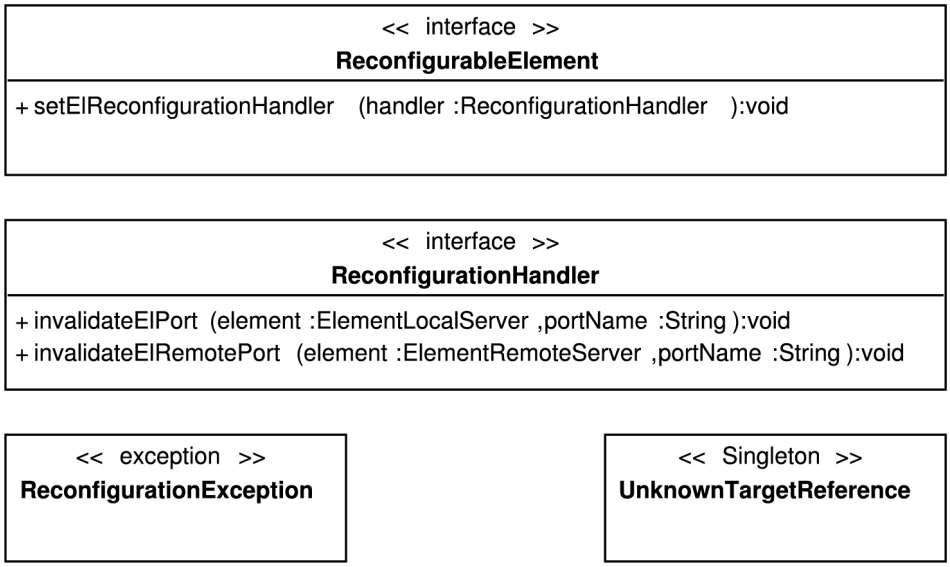


Figure 52: Connector runtime model entities for reconfiguration support

elements as well as propagate reconfiguration requests to higher levels of connector architecture.

The entities of the connector runtime responsible for connector instantiation and binding will only implement the ReconfigurationHandler interface, to provide the top level architecture elements representing connector units with a means to trigger reconfiguration on the level of a connector instance. In case of a connector unit bound to a client component, it allows to rebind a component to a new local reference provided by the unit, in case of a connector unit bound to a server component, it allows propagating changes in remote references to other connector units in different address spaces.

The base implementations of connector architecture entities such as primitive and composite elements need to be adapted to implement the reconfiguration algorithm. The following paragraphs outline the basic role of different types of runtime entities, responsible for maintaining connector architecture at runtime, in the reconfiguration mechanism. More detailed description of the reconfiguration algorithm, including pseudo-code of methods implementing element control interfaces, is provided in the next section.

Primitive Element

When a primitive element detects a change in configuration that influences some of its provided ports (this may be because the element received a new binding to its required port or due to an external request) it uses the ReconfigurationHandler callback interface provided by its parent and signals the change using invalidateElPort method, passing a reference to itself and the name of the port that is to be invalidated as method arguments. The parent which handles the reconfiguration request is responsible for performing a lookup on the element's provided port (so that the element can provide a new reference to the port) and for making the new reference available to all users of the invalidated provided port.

Composite Element

The behavior of a composite element is a more complex. A composite element typically delegates its required and provided ports to ports of its child elements. Therefore requests to bind a composite element's required port to another provided port are passed down to child elements which actually bind to the provided port. Analogously, requests to lookup a reference to a composite element's provided port are passed down to child elements which provide the actual reference.

As for configuration changes, a composite element may request reconfiguration either due to an external request, or on behalf of one of its child elements.

If the required ports of a composite element are delegated to its child elements, a request to bind the port to another provided port can trigger reconfiguration requests from one of the child elements.

If a reconfiguration request from a child element invalidates a port that is part of a binding between two child elements, the composite element queries the signaling element for a new reference to the invalidated port and provides the new reference to all elements bound to the invalidated port.

If the invalidated port is delegated to the composite element's provided port, the composite element propagates the reconfiguration request to its parent using its own ReconfigurationHandler callback.

A composite element may decide to use facades on its provided and required ports. In case of required ports it may not propagate bind requests to child elements, while in case of provided ports it may not propagate reconfiguration requests from a child element to its parent.

If a composite element represents a connector unit, it has no parent element. However, it still needs a reconfiguration handler to allow propagating changes across address space boundaries. In this case, the reconfiguration handler is provided by a connector runtime responsible for a connector instance.

Connector runtime

Connector runtime manages connectors and connector units and provides reconfiguration handlers to top-level connector elements which represent the units. If a connector runtime receives a reconfiguration request from a connector unit, it handles it in a way similar to that of a composite element, except on the level of connectors and components.

If a reconfiguration signaled by a connector unit occurred on unit's provided port, which is bound to a component's required port, the connector runtime queries the connector unit for a new reference to the invalidated port and provides this reference to the component that is bound to the unit.

If the reconfiguration request occurred on unit's remote port, the connector runtime queries the port for a new set of remote references and provides these references to the units that are bound to the invalidated remote port.

5.5.2 Reconfiguration algorithm

The reconfiguration algorithm is basically executed in a distributed fashion by all runtime entities associated with connector architecture. Because the algorithm operates on a

hierarchical structure with strict encapsulation, each participant has only local knowledge and a link to its parent entity to work with.

The link is realized through ReconfigurationHandler interface, which is implemented by all non-leaf entities of the connector architecture, i.e. composite elements and connector runtime, and during instantiation provided to all non-root entities, i.e. composite and primitive elements. When the reconfiguration process needs to traverse the connector architecture upwards (along the delegated ports), the respective connector element uses the interface to initiate the reconfiguration process in its parent entity. In the case of our connector model, the ReconfigurationHandler interface contains two methods that serve for invalidating provided and remote server ports of the element initiating the reconfiguration process.

When disabling an optional feature provided by an element, the element remains in the connector architecture but is excluded from a call path passing through its ports. As described in Section 5.5.1, when an element wants to be excluded from the call chain, it provides target reference associated with its required port as a reference to its provided port. This constitutes an internal dependency between the ports which the reconfiguration process needs to be able to traverse. Since this dependency is not allowed between all port types, we enumerate the allowed cases and introduce the notion of a pass-through port.

An element port is considered pass-through, iff one of the following holds:

a) the port is provided and is associated with exactly one required port of the same element, and if the reference to the port is looked up, the target of the associated required port is returned instead,

b) the port is provided and is associated with exactly one remote client port of the same element, and if the reference to the port is looked up, a particular target from the reference bundle of the associated remote port is returned instead,

c) the port is remote server and is associated with one or more required ports of the same element, and if the reference to the port is looked up, the returned reference bundle contains also the targets of the associated required ports.

Pass-through target is a target object of a required port that is associated with a pass-through port and to which the invocations on the pass-through port are delegated.

The reconfiguration process can be initiated at any non-root entity of the connector architecture, whenever a connector element needs to change a reference to an object implementing its provided or remote server port. This can happen either when an element's required or remote client port associated with a pass-through port is bound to a new target, or when an element needs to change the reference in response to an external request.

The implementation of each element must be aware of the potential internal dependencies between the pass-through ports and their pass-through targets. Whenever a change occurs in an element that changes the internal dependencies, the element must initiate the reconfiguration process by notifying its parent entity about its provided and remote server ports that are no longer valid.

The implementation of the reconfiguration algorithm is spread among the entities of the connector architecture. These entities implement methods declared in the following

interfaces and their implementation is different for root, intermediary, and leaf entities in the connector architecture hierarchy.

Element interfaces

All non-root entities, i.e. both composite and primitive elements, implement one or more of the four element control interfaces (see Figure 40 for reference):

- ElementLocalServer, ElementLocalClient,
- ElementRemoteServer, ElementRemoteClient.

The implementation of these interfaces in a composite element differs from that of a primitive element.

Reconfiguration interface

Non-leaf entities, i.e. composite elements and connector runtime implement the ReconfigurationHandler interface so that they can service reconfiguration requests from child entities. The implementation of this interface in a composite element differs from that of connector runtime.

Lookup Port operation

Serves for looking up a local reference to an object implementing a particular provided port. Serviced by the lookupElPort method of the ElementLocalServer interface. The providedPortName argument determines the name of the port to look up.

Method lookupElPort*, primitive element implementation:*

The implementation for a primitive element checks if the named port is pass-through and if yes, the target object bound to the associated required port is returned, otherwise a reference to an internal (possibly this) object implementing the named port is returned. If the associated port has not been yet bound, its target object reference will contain a special value UnknownTargetReference which, when returned to the caller, indicates that the named port does yet not have a target.

```
providedPort = localProvidedPorts [providedPortName]

if (providedPort is pass-through):
    targetRequiredPort = providedPort.passthroughTargetPort
    return targetRequiredPort.targetObject
else:
    return providedPort.targetObject
```

Method lookupElPort*, composite element:*

The implementation for a composite element differs from the implementation for a primitive element in that it supports delegation of its provided ports to provided ports of its sub-elements.

```
providedPort = localProvidedPorts [providedPortName]
```

```
if (providedPort is pass-through):
    targetRequiredPort = providedPort.passthroughTargetPort
    return targetRequiredPort.targetObject
else if (providedPort is delegated):
    { targetElement, targetPortName } = providedPort.delegationTarget
    return targetElement.lookupElPort (targetPortName)
else:
    return providedPort.targetObject
```

Lookup Remote Port operation

Serves for looking up a bundle of remote references to objects providing entry-points to the implementation of a particular remote server port. Each reference in the bundle is associated with a particular access scheme. Serviced by the lookupElRemotePort method of the ElementRemoteServer interface. The serverPortName argument determines the name of the port to look up.

Method lookupElRemotePort*, primitive element implementation:*

Creates a new reference bundle for storing remote references. If the named port is pass-through, adds references to pass-through targets of the associated required ports to the bundle using a "local" reference scheme. Then it adds references to all internal skeleton objects with their respective reference schemes to the reference bundle. Finally it returns the resulting reference bundle.

```
serverPort = remoteServerPorts [serverPortName]
referenceBundle = new ReferenceBundle ()

if (serverPort is pass-through):
    localReferenceList = new LocalReferenceList ()
    foreach (targetRequiredPort in serverPort.passthroughTargetPorts):
        localReferenceList.append (targetRequiredPort.targetObject)
    referenceBundle.add ("local", localReferenceList)

foreach (skeleton in serverPort.targetSkeletons):
    referenceBundle.add (skeleton.schema, skeleton.reference)

return referenceBundle
```

Method lookupElRemotePort*, composite element implementation:*

Similar to the lookupElPort, the implementation of the lookupElRemotePort method in a composite element also handles delegation of its remote ports to remote ports of its sub-elements. In contrast to provided ports, a remote port can be delegated to multiple elements.

```
serverPort = remoteServerPorts [portName]
referenceBundle = new ReferenceBundle ()

if (serverPort is pass-through):
    localReferenceList = new LocalReferenceList ()
```

```
        foreach (targetRequiredPort in serverPort.passthroughTargetPorts):
            localReferenceList.append (targetRequiredPort.targetObject)

        referenceBundle.add ("local", localReferenceList)

    if (serverPort is delegated):
        foreach (delegation in elementArchitecture.getDelegations ()):
            if (delegation.origin.portName == serverPort.portName):
                { targetElement, targetPortName } = delegation.target
                bundlePart = targetElement.lookupElRemotePort (targetPortName)
                referenceBundle.add (bundlePart)

    foreach (skeleton in serverPort.targetSkeletons):
        referenceBundle.add (skeleton.schema, skeleton.reference)

    return referenceBundle
```

Bind Port operation

Serves for binding a particular required port to a provided port. Serviced by the bindElPort method of the ElementLocalClient interface. The requiredPortName argument determines the required port to bind, while the targetObject argument contains a local reference to an object implementing the port.

Method bindElPort*, primitive element implementation:*

Checks whether reconfiguration is in progress for the given port, because attempt to bind a port for which reconfiguration is already in progress indicates invalid connector architecture and results in termination of the algorithm. If the check passes, sets a flag indicating that reconfiguration is in progress for this port, stores the target object reference, invalidates all pass-through provided and remote ports depending on the given port, and clears the port's reconfiguration-in-progress flag before returning.

```
requiredPort = localRequiredPorts [requiredPortName]

if (requiredPort.reconfigurationInProgress == true):
    throw ReconfigurationException ("Reconfiguration cycle detected")
else:
    requiredPort.reconfigurationInProgress = true
    requiredPort.targetObject = targetObject

    foreach (providedPort in requiredPort.providedPassthroughPorts):
        reconfigurationHandler.invalidateElPort (this, providedPort.portName)

    foreach (remotePort in requiredPort.remotePassthroughPorts):
        reconfigurationHandler.invalidateElRemotePort (this, remotePort.portName)

    requiredPort.reconfigurationInProgress = false
```

Method bindElPort*, composite element implementation:*

In addition to the functionality of the primitive element implementation, the composite element implementation supports recursive passing of the target object reference down the connector architecture hierarchy to the sub-elements that subsume their required ports to the containing composite element.

```
requiredPort = localRequiredPorts [requiredPortName]

if (requiredPort.reconfigurationInProgress == true):
    throw ReconfigurationException ("Reconfiguration cycle detected")
else:
    requiredPort.reconfigurationInProgress = true
    requiredPort.targetObject = targetObject

    foreach (subsumption in elementArchitecture.getSubsumptions ()):
        if (subsumption.target.portName == requiredPort.portName):
            { subsumingElement, subsumingPortName } = subsumption.origin
            subsumingElement.bindElPort (subsumingPort, targetObject)

    foreach (providedPort in requiredPort.providedPassthroughPorts):
        reconfigurationHandler.invalidateElPort (this, providedPort.portName)

    foreach (remotePort in requiredPort.remotePassthroughPorts):
        reconfigurationHandler.invalidateElRemotePort (this, remotePort.portName)

    requiredPort.reconfigurationInProgress = false
```

Bind Remote Port operation

Serves for binding a particular remote client port to a remote server port. Serviced by the bindElRemotePort method of the ElementRemoteClient interface. The clientPortName argument determines the client port to bind, while the targetReferences argument contains a bundle of remote references to objects that provide entry point to the implementation of the port.

Method bindElRemotePort*, primitive element implementation:*

Stores the bundle for remote references and invalidates all provided pass-through ports depending on the given remote client port.

```
clientPort = remoteClientPorts [clientPortName]
clientPort.targetReferences = targetReferences

foreach (providedPort in clientPort.providedPassthroughPorts):
    reconfigurationHandler.invalidateElPort (this, providedPort.portName)
```

Method bindElRemotePort*, composite element implementation:*

In addition to the functionality of the primitive component implementation, the composite element implementation supports recursive passing of the target reference bundle down the connector architecture hierarchy to the sub-elements that subsume their remote client ports to the containing composite element.

```
clientPort = remoteClientPorts [clientPortName]
clientPort.targetReferences = targetReferences

foreach (subsumption in elementArchitecture.getSubsumptions ()):
    if (subsumption.target.portName == clientPort.portName):
        { subsumingElement, subsumingPortName } = subsumption.origin
        subsumingElement.bindElRemotePort (subsumingPortName, targetReferences)

foreach (providedPort in clientPort.providedPassthroughPorts):
    reconfigurationHandler.invalidateElPort (this, providedPort.portName)
```

Invalidate Port operation

Serves for invalidating a reference to an object implementing a particular provided port. Used by connector elements to indicate that the provided port should be queried for a new reference. Serviced by the invalidateElPort method of the ReconfigurationHandler interface, which is implemented by the composite elements and the dock connector manager. Each connector element instance is supplied with an implementation of this interface by its creator. The serverElement argument identifies the element initiating the reconfiguration, while the serverPortName argument identifies the provided port that is being invalidated.

Method invalidateElPort*, composite element implementation:*

Obtains a new reference to a local object implementing the invalidated port. Rebinds all required ports of other elements bound to the invalidated provided port, supplying the newly obtained reference as the target object. Provided ports of the composite element that are delegated to the invalidated port are also invalidated using the reconfiguration handler supplied by the creator of the element.

```
targetObject = serverElement.lookupElPort (serverPortName)

foreach (binding in elementArchitecture.getBindings ()):
    if (binding.server == { serverElement, serverPortName }):
        { clientElement, clientPortName } = binding.client
        clientElement.bindElPort (clientPortName, targetObject)

foreach (delegation in elementArchitecture.getDelegations ()):
    if (delegation.target == { serverElement, serverPortName }):
        delegatingPortName = delegation.origin.portName
        reconfigurationHandler.invalidateElPort (this, delegatingPortName)
```

Method invalidateElPort*, dock connector manager implementation:*

Top level reconfiguration handler. Associated with a particular connector instance. Obtains a new reference to a local object implementing the invalidated port. Rebinds all component interfaces bound to the invalidated provided port, supplying the newly obtained reference as the target object.

```
targetObject = serverElement.lookupElPort (serverPortName)
```

```
foreach (binding in connectorArchitecture.getComponentBindings ()):
    if (binding.server == { serverElement, serverPortName }):
        { clientComponent, interfaceName } = binding.client
        clientComponent.bindInterface (interfaceName, targetObject)
```

Invalidate Remote Port operation

Serves for invalidating a bundle of remote references to objects providing an entry point to the implementation of a particular remote server port. Used by connector elements to indicate that the remote server port should be queried for a new reference bundle. Serviced by the invalidateElRemotePort method of the ReconfigurationHandler interface, which is implemented by the composite elements and the dock connector manager. Each connector element instance is supplied with an implementation of this interface by its creator. The serverElement argument identifies the element initiating the reconfiguration, while the serverPortName argument identifies the remote server port that is being invalidated.

Method invalidateElRemotePort*, composite element implementation:*

Invalidates all remote server ports of the composite element that are delegated to the invalidated remote server port using the reconfiguration handler supplied by the creator of the element.

```
foreach (delegation in elementArchitecture.getDelegations ()):
    if (delegation.target == { serverElement, serverPortName }):
        delegatingPortName = delegation.origin.portName
        reconfigurationHandler.invalidateElRemotePort (this, delegatingPortName)
```

Method invalidateElRemotePort*, dock connector manager implementation:*

Top level reconfiguration handler. Associated with a particular connector instance. Initiates rebinding of connector units for bindings with remote client ports bound to the invalidated remote server port.

```
foreach (binding in connectorArchitecture.getUnitBindings ()):
    if (binding.server == { serverElement, serverPortName }):
        rebindConnectorUnits (connectorInstance, binding)
```

Rebind Connector Units operation

Serves for rebinding the remote client ports to remote server ports among connector units. Serviced by the rebindConnectorUnits method implemented by the dock connector manager (connector architecture runtime). The connectorInstanceName argument determines the instance of a connector and the remoteBindingName determines the binding in the connector architecture that needs to have its units rebound.

Method rebindConnectorUnits*, dock connector manager implementation:*

Collects remote references from all remote server ports participating in a particular binding and merges them to one remote reference bundle. Binds all remote client ports participating in the binding to the newly obtained reference bundle.

```
referenceBundle = new ReferenceBundle ()

foreach (remote port participating in the remote binding remoteBindingName):
    unitReferenceBundle = unitElement.lookupElRemotePort (remotePortName)
    referenceBundle.add (unitReferenceBundle)

foreach (remote port participating in the remote binding remoteBindingName):
    unitElement.bindElRemotePort (remotePortName, referenceBundle)
```

Bind Component Interface operation

Serves for binding a particular component interface to a provided connector element port implementing the interface. Serviced by a method specific to a particular component-model. For demonstration purposes, we assume this operation to be serviced by bind-Interface method of a closely unspecified component runtime interface. The interface-Name argument determines the interface to bind, while the targetObject argument contains a local reference to an object implementing the interface.

Additional discussion

The reconfiguration algorithm propagates information about changes in the connector implementation along the edges of a dependency graph, its vertices and edges representing element ports and the bindings between them, including the internal associations between pass-through ports and pass-through targets. The propagation of changes corresponds to a breadth-first traversal of the dependency graph, starting at the vertex that initiated the change. Since the information required to explicitly construct the dependency graph is not available due to encapsulation, the situation is more complex in our case.

The algorithm has to traverse the dependency graph indirectly, using the runtime structures designed to reflect the connector model. The dependency information is distributed over the connector architecture hierarchy, where each non-leaf entity in the connector hierarchy only has dependency information concerning its direct successors (child composite and primitive elements).

This distribution of dependency information is a consequence of the hierarchical connector model, which limits a connector element's awareness of the surrounding environment. This in turn eases composition of connector elements with limited but clearly defined function.

During execution, the algorithm performs two main operations. First, it propagates change notifications to non-leaf elements at upper levels of the connector architecture hierarchy. Second, it distributes updated information to non-root elements at lower levels of the hierarchy. In case of connector architecture, the information that needs to be distributed comprises references to implementations of element ports. The change notification issued through the ReconfigurationHandler interface tells the receiver that a reference to a specific port of a specific element has been invalidated, which means that the element has to be queried for a new reference to that port.

The propagation of notifications to the upper levels is necessary because an element does not have information about dependencies beyond its boundary and does not know anything about elements that depend on it. Therefore, an element notifies its parent element about change whenever it affects its provided port.

The parent element receiving the notification uses its local binding information to find its own ports (delegation) and ports of the child elements (binding) that depend on the provided port of the signaling element. If any required port of the child elements depends on the invalidated port, the parent element obtains an updated reference to the object implementing the invalidated port and provides it to all dependent ports. If any of its own ports are affected by the change, it propagates the change notification to its own parent element by invalidating the dependent ports.

The distribution of the updated object reference to the dependent elements at a lower level may trigger additional change notifications originating either in these elements or elements at even lower levels. This is because whenever an element receives a target reference to a provided port required by a pass-through target, it may affect the associated pass-through port.

5.5.3 Algorithm Termination

Given the recursive nature of the algorithm, an obvious question is whether it always terminates. We show that the answer is yes, even when an implementation of a connector architecture is invalid, in which case the connector may be left in defunct state. In proving that the algorithm always terminates, we will examine the reasons for the algorithm not to terminate and show that such situation cannot happen. We believe that the use of informal language does not affect the correctness of the proof.

As discussed in the previous section, even though the reconfiguration algorithm operates with the connector architecture, it uses it only for indirect traversal of a graph of dependencies between element ports. Because a connector as a whole can only expose local ports of its elements to the outside, these ports cannot be depended on by other element ports from lower levels of the architecture. Consequently, the traversal of the dependency graph must reach these top-level ports and stop there, unless there is a cycle in the dependency graph at lower levels of the connector architecture.

An implementation of connector architecture with cycles in its port dependency graph is clearly invalid, but this cannot be verified in advance, because the complete dependency information is not explicitly available. Additionally, the implementation of connector architecture may have been initially valid and became invalid only after changes at runtime. The port dependency graph of connector architecture must be therefore assumed to contain cycles and the algorithm must detect these cycles at runtime and terminate if a cycle is detected.

Since cycles in the dependency graph will inevitably lead to cycles in the call graph of the methods implementing the algorithm, then detecting cycles in the method call graph allows us to detect cycles in the dependency graph as well. The call graph depicted in Figure 53 was derived from the pseudo-code of methods described in Section 5.5.2. The nodes of the graph correspond to methods, while the edges between nodes correspond to method calls. The edges are oriented and lead from the caller to the callee.

The graph in Figure 53 captures all variants (depending on the position of the implementing entity in connector architecture) of the methods implementing the reconfiguration algorithm, which makes it unnecessarily complex. Since the call graphs for the different method variants are very similar and the proof only needs to analyze the cycles, the parts of the graph that do not provide additional information can be merged, which are precisely the parts that the differentiate between method variants. The simplified graph depicted in Figure 54 may be a bit more benevolent (in terms of allowed

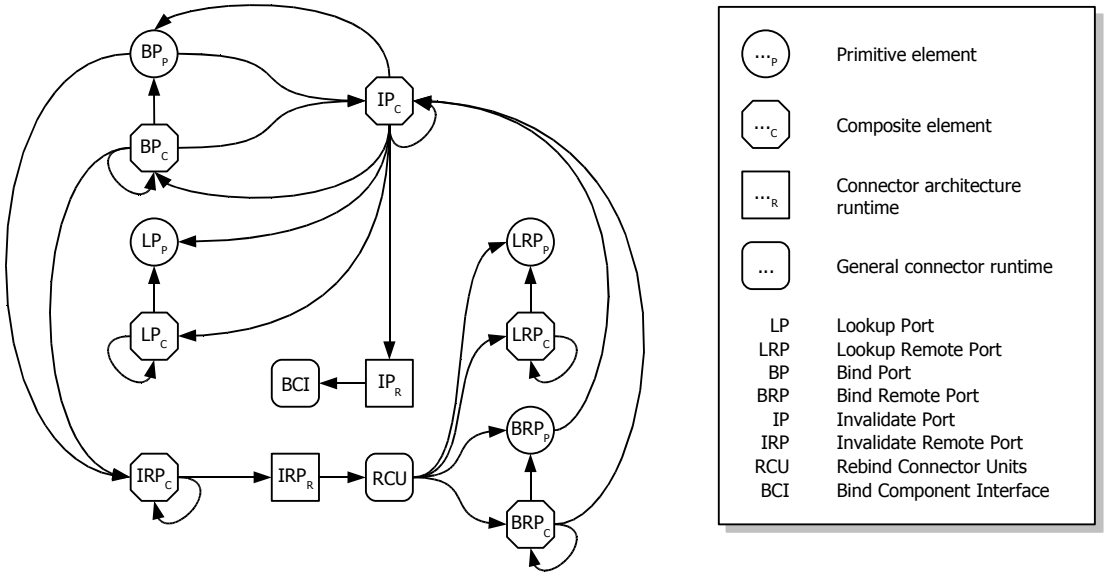


Figure 53: Reconfiguration call graph

calls) than the graph in Figure 53, but if the proof can be made for the simplified graph, it will also hold for the original graph.

The graph in Figure 54 contains a number of cycles. We first analyze the trivial cycles associated with the nodes labeled LP, LRP, BP, BRP, IP, and IRP. In case of the LP, LRP, BP, and BRP nodes, these cycles correspond to delegation of the respective Lookup Port, Lookup Remote Port, Bind Port, and Bind Remote Port operations from the parent elements to the child elements, down the connector architecture hierarchy. The recursion of these operations is limited by the depth of the connector architecture hierarchy, because eventually the methods must reach a primitive element, which is a leaf entity of the connector architecture. In case of the IP and IRP nodes, the cycles correspond to the propagation of the Invalidate Port and Invalidate Remote Port operations from a connector element to its parent entity (another connector element or connector runtime). The recursion is again limited by the depth of the connector architecture hierarchy, but in this case these operations must eventually reach to root entity (connector runtime).

The cycles associated with the BP, BRP, IP, and IRP nodes can be also part of other, non-trivial, cycles. One class of them can be derived from the IP-BP cycle, which corresponds to the distribution of a new port reference to other elements. The Bind Port operation traverses the connector architecture downwards, while the Invalidate Port operation upwards. Because the number of required ports in a connector element is finite, then if the dependency graph contains a cycle, the Bind Port operation will inevitably get called again for the same port before the previous call finished. Since the Bind Port operation has to be part of every such cycle, the operation is guarded against reentrant invocation for the same port using a per port flag indicating that reconfiguration is already in progress for the port. The use of the flag corresponds to marking of the path during traversal to detect cycles. Therefore if a Bind Port operation finds the flag set already set in the port it was called for, there must be a cycle in the dependency graph and the algorithm is terminated.

The other non-trivial class of cycles can be derived from the IP-BP-IRP-RCU-BRP cycle. These cycles can be longer and more complex than in the previous case, but if the port dependency graph does not contain a cycle, they are also limited by the size of the connector architecture which is finite. If the port dependency graph contains a cycle, it will

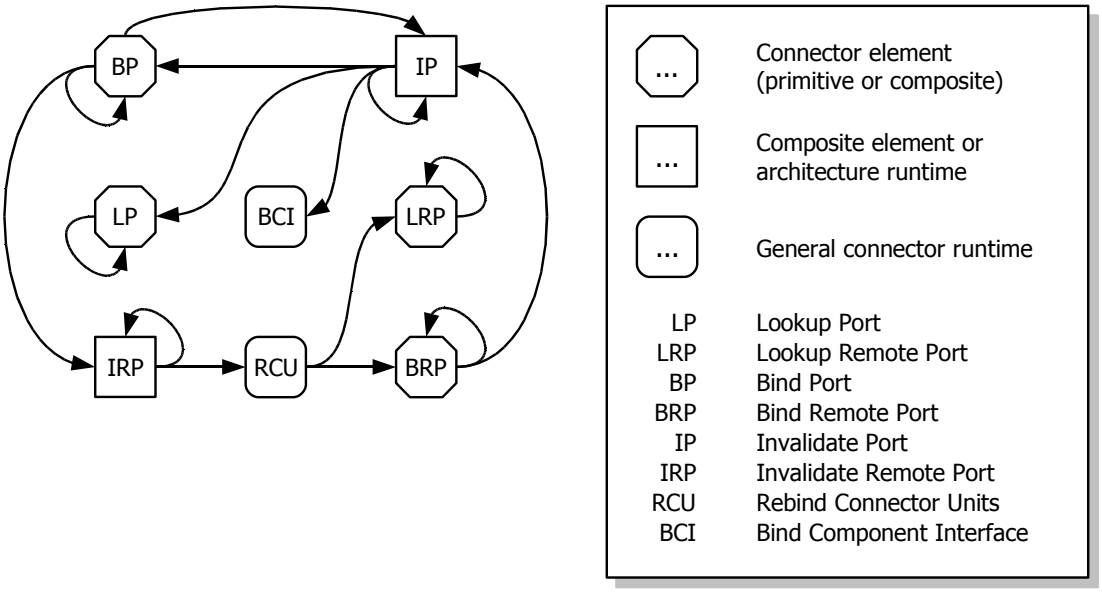


Figure 54: Simplified reconfiguration call graph

be detected by the Bind Port operation, because the calls must always pass through the BP node, thereby locking part of the traversal path against recursion. Other than that, the calls can divert from the main IP-BP-IRP-RCU-BRP cycle to the trivial cycles associated with IRP, BRP, and IP nodes, but there they can only traverse the connector architecture upwards or downwards, eventually reaching either the root or a leaf of the hierarchy.

Because no other cycles are possible in the method call graph, the algorithm always terminates in finite number of steps because the connector architecture is finite. In case of an invalid implementation of connector architecture, the algorithm is terminated prematurely when a cycle in the port dependency graph is detected.

Chapter 6

Heterogeneous Deployment

Application deployment is a process of transporting an application, a final product of development and assembly, to the target execution environment and launching it to obtain a running instance of the application. The execution environment can be anything from a single computer with a specific operating system, to a set of computers connected via network, each with its own hardware architecture, operating system, and other software infrastructure.

For typical applications, the process is straightforward and consists mainly of downloading or otherwise obtaining an application package, installing it onto the target system and launching it. If an application requires specific libraries or other applications to be present on the target system, the installation tool must resolve these dependencies prior to installation. When all the requirements are satisfied, an application can be started. In case of interactive applications, this is typically done by users; in case of non-interactive applications this is done by an operating system, provided that the application is properly registered with the system. This is a common desktop computing experience, which may be more or less painful, depending on a particular operating system. The important fact is that an application is typically started by launching an executable file, which resides on a local file system along with all other application files.

With distributed component-based applications, the process becomes more complex because an application may be composed of components that need to be running on different nodes. This requires installing application components onto a number of selected nodes and launching the component runtime on those nodes to allow execution of components assigned to a particular node. In the most complex case, there may be multiple computational nodes on a single physical computer, the nodes can be connected with each other either directly or indirectly, each of the nodes may run different operating system and support different communication paradigms, and the application may consist of components implemented in different component systems deployed on multiple nodes.

In the context of this thesis, application deployment is used as an integration platform, where we first address the issues related to deployment of heterogeneous component-based applications (covered by Sections 6.1, 6.2, 6.3, and 6.4) and then use it for transparent integration of performance instrumentation process into life cycle of component-based applications (covered by Section 6.5).

Note: Sections 6.1, 6.2, and parts of Sections 6.3, and 6.4 were taken verbatim from [BB04] and adapted for use in the text of the thesis. Sections6.3, and 6.4 were updated to better reflect the current state of the research, but the results published in [BB05] remain valid and are not influenced by the changes or occasional different wording used to improve readability of the original text.

6.1 Deploying Component-based Applications

Application deployment is an issue which is common to most component systems supporting distributed applications. Consequently, most implementations of component runtimes attempt to address the issue in some way, but the differences between various component models have made it difficult to arrive at a common solution. The differences comprise mainly component packaging and transport, communication middleware, hierarchical composition, component instantiation, and life cycle management. As a result, integration and maintenance of applications based on different component systems is very difficult.

The deployment process generally consists of several steps which have to be performed to launch an application. The process is typically specific to a particular component technology. In some cases, even applications based on a standardized component technology have to be deployed using tools and processes specific to a particular vendor of the technology.

A specification [OMG06c] issued by the Object Management Group provides a conceptual framework for industrial standardization of deployment and configuration of component-based distributed applications. However, the specification does not explicitly support heterogeneous component-based applications. In the context of our research in the area of component-based software development, we are working on designing and developing a generic deployment infrastructure which will provide support for heterogeneous component-based applications and still be compatible with the deployment process defined in the OMG specification. To demonstrate the problems associated with deployment of heterogeneous component applications as well as the feasibility of our approach, we consider deployment of applications assembled from components based on the SOFA [PBJ98, OWC07b], Fractal [OWC07a], and EJB [SUN03] component systems.

One of the main problems inherent to deployment of heterogeneous component applications is related to interconnection of components from different component models. The problem arises mainly due to

- different middleware platforms used by the component systems to achieve distribution, and
- different ways of instantiating components and accessing their interfaces.

Of the three mentioned component models, SOFA offers the most freedom in the choice of middleware, as it has native support for software connectors, which allow using arbitrary middleware for communication in the context of a particular communication style. Fractal, on the other hand, supports distribution with its own Fractal RMI middleware based on serialization defined by SUN RMI [SUN01]. However, the middleware is not compatible with the classic SUN RMI. Finally, EJB uses SUN RMI to achieve distribution.

Regarding the component instantiation mechanisms, the SOFA and the Fractal component models are quite similar. Both employ the concept of factory (component builder in SOFA, generic factory in Fractal) for creating component instances, yet they differ substantially in the way a component structure is described. The SOFA model describes the structure statically, using SOFA-specific ADL called Component Definition Language. In Fractal, the description of the structure is dynamic, passed as a parameter to the generic factory. However, still there is still a distinction between components' code and code for instantiating and binding of components.

The EJB component model, on the other hand, bears very little similarity to either of the discussed models. The EJB component model supports four different kinds of components referred to as beans:

a) entity beans, stateful, the state is persistent and usually stored in a database,

b) stateful session beans, the state of which is preserved for the duration of a session,

c) stateless session beans, which are quite similar to libraries, and

d) message-driven beans, which are similar to stateless session beans, except they lack the classic business interface, and instead process incoming requests in a message loop.

Every component has a business interface and a home interface. The home interface of a bean is used to instantiate components of a specific kind, and in case of entity beans, restore component state from a persistent store. Bean home interfaces can be obtained through naming service. The actual instantiation is performed from components' and client's code. Prior to any request to the naming service, a bean has to be first deployed into an EJB container, using implementation-specific deployment tools. Unlike the SOFA and the Fractal models, EJB does not support component nesting.

To overcome the differences between these models, we have decided to use software connectors to facilitate the bridging between these technologies because the concept is very flexible and, because of its compositional character, generally applicable. Software connectors encapsulate all communication among components and are typically responsible for

1. distribution (employing a communication middleware),
2. adaptation (hiding changes in method names, and order of arguments, or performing more complex transformation), and
3. additional services related to non-functional requirements on a particular connection (such as encryption, performance and behavior monitoring, rate limiting).

Although the connector-based approach to bridging differences between component models appears to be very promising, the OMG specification does not natively support connectors for component interconnection and expects applications components to be directly connected. Therefore the concept of connectors has to be first integrated into the specification of the deployment process defined by OMG. We intend to preserve compatibility with the original specification; therefore to minimize the impact of integrating connectors into the specification of the deployment process, we attempt to map the concept of connectors onto concepts already present in the specification.

With regard to the discussion above, in the following sections we aim to present how to

1. integrate connectors into the specification of deployment process defined by OMG while preserving compatibility with the original specification, and
2. use connectors to overcome the differences between the example component models to show the feasibility of the connector-based approach.

6.2 Overview of the OMG D&C Specification

The specification [OMG06c] describes and standardizes the relations between three major concepts. The first is the concept of an application assembled of other components, the application itself being a component. In this context, components are considered to be reusable units of work that are independently useful. The second is the concept of target environment, referred to as domain, which provides computational resources for execution of component-based applications. The third is the concept of deployment as a process which takes a component-based application and a target environment as an input and produces an instance of the application running in the target environment on its output.

Given enough information about the application and the target environment, the deployment process is expected to be reasonably generic, especially at higher levels of abstraction. The required information is made available to the process in form of detailed description with a standardized data model. To allow for specialization at lower levels of abstraction, the specification is compliant with the Model Driven Architecture (MDA) approach [OMG01], also defined by OMG. The core of the specification defines a set of concepts and classes relevant for the implementation of the specification, which form a platform independent model (PIM) of the specification. This model can be then transformed to a platform specific model (PSM), which can capture the specifics of a particular component technology, programming language, or information formatting technology.

The component model defined by the core specification is explicitly independent of distributed component technologies such as CORBA Component Model (CCM) [OMG06b] or Enterprise Java Beans (EJB) [SUN03]. Components can be implemented either directly (a monolithic implementation), or as an assembly of other components. This provides support for hierarchical composition which allows capturing the logical structure of an application and configuration of component assemblies. Ultimately, though, every application can be decomposed into a set of components with monolithic implementation, which is the form required for deployment.

The target environment, a domain, consists of nodes, interconnects and bridges. Of these, only the nodes provide computational capabilities and resources, while interconnects serve for grouping nodes that are able to communicate directly within a domain. A situation where the nodes cannot, for some reason (e.g. a firewall, an application proxy), communicate directly is modeled by grouping the nodes in different interconnects. Bridges are then used to facilitate communication between nodes in different interconnects.

The deployment process consists of the following five stages: *installation*, *configuration*, *planning*, *preparation*, and *launch*. Prior to deployment, the application must be packaged and made available by the producer. The package has to contain all relevant metadata describing the application as well as code and data artifacts required to run the application.

To minimize the amount of interdependencies and to lower the overall complexity of the platform independent model, the specification defines two dimensions for segmenting the model into modules. The first dimension provides a distinction between a data model of the descriptive information and a management model of runtime entities, which process the information. The second dimension takes into account the role of the models in the deployment process, and distinguishes among component, target, and execution models.

Since giving a complete overview of the whole specification is beyond the scope of this thesis, we have selected the main parts required to provide context for the presented work. Of the modules mentioned earlier, we provide detailed description only for the component and execution data models, along with a brief overview of the deployment process.

6.2.1 Component Data Model

The component data and management models are mainly concerned with description and manipulation of component software packages. The description specifies requirements that have to be satisfied for successful deployment, most of which are independent of a particular target system. Both the application metadata and code artifacts are expected to be stored and configured in a deployment repository during the installation and configuration stages of the deployment. The information in the repository will be accessed and used during the planning, and preparation stages.

Figure 55 shows a high level overview of the component data model. The key concept is a component package, which contains the configuration and implementation of a component. If a component has multiple implementations, the configuration should specify selection requirements, which influence deployment decisions by matching the requirements to capabilities of individual implementations.

Each component package realizes a component interface, which is implemented by possibly multiple component implementations. Figure 56 shows a detailed view of a component interface description. A component interface is a collection of ports, which can represent endpoints in connections among components. A collection of properties carries component interface configuration.

As shown in Figure 55, an implementation of a component can be either monolithic, or represented by an assembly of other components. In case of monolithic implementation, the description of the implementation consists of a list of implementation artifacts that make up the implementation. The artifacts can depend on each other and can be associated with a collection of deployment requirements and execution parameters; however this is not shown in the figure to reduce clutter. The requirements have to be satisfied before an implementation artifact can be deployed on a target node.

A component implementation that is not monolithic is defined as an assembly of other components. Figure 57 shows a detailed view of a component assembly description. An assembly describes instances of subcomponents and connections among them. A subcomponent instance can reference a component package both directly and indirectly. Indirect package reference contains a specification of component interface the package has to realize and is expected to be resolved before deployment.

A set of selection requirements is part of an instance description and serves in choosing an implementation when a component package contains multiple implementations. Since the configuration of an assembly needs to be delegated to the configuration of its subcomponents, the description of an assembly contains a mapping of its configuration properties to configuration properties of its component instances.

The instances of components inside the assemblies are connected using connections. A connection description contains a set of endpoints and deployment requirements for the connection. There can be three kinds of endpoints: a port of subcomponent's component interface, an external port of the assembly, or an external reference.

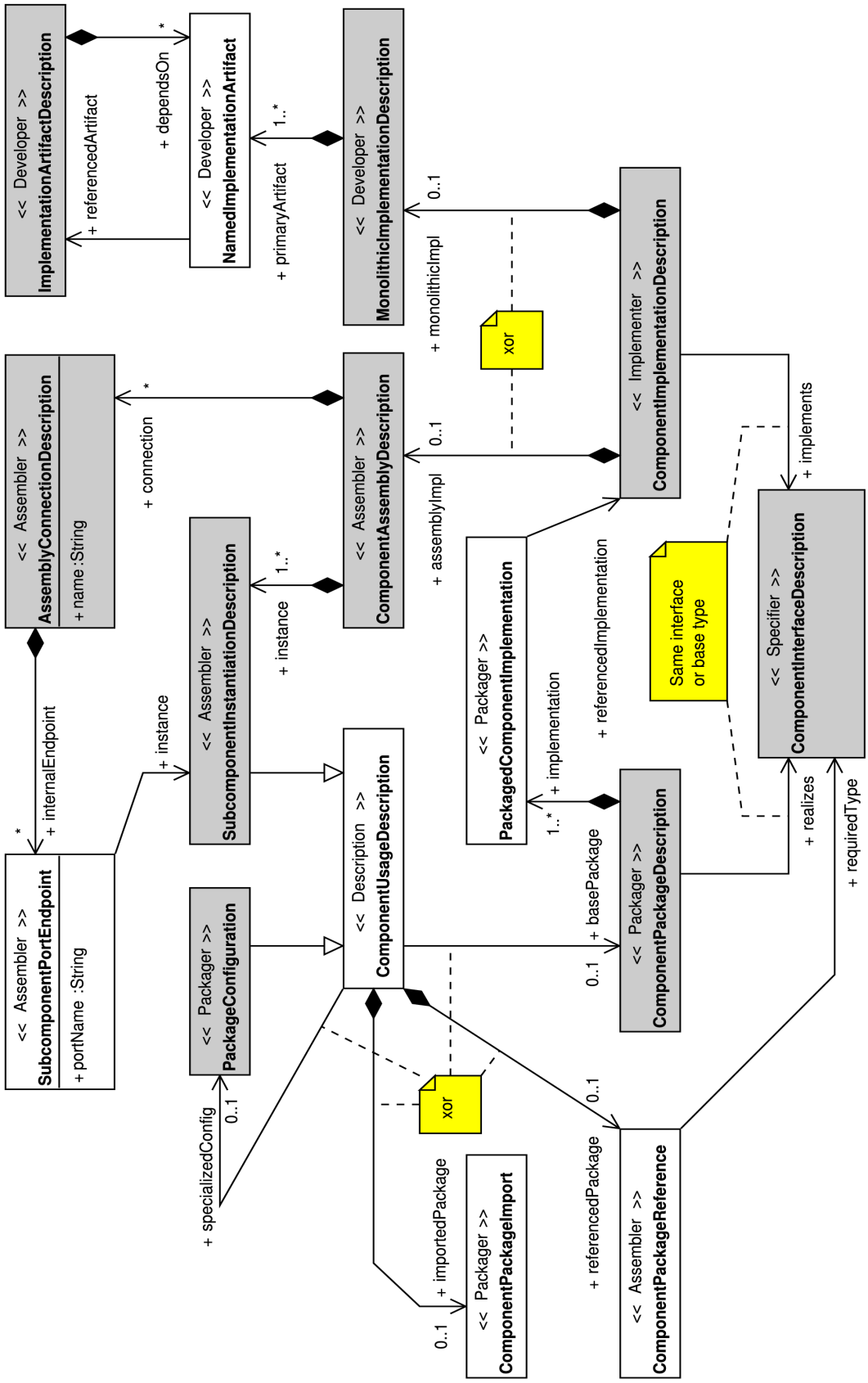


Figure 55: Overview of the component data model

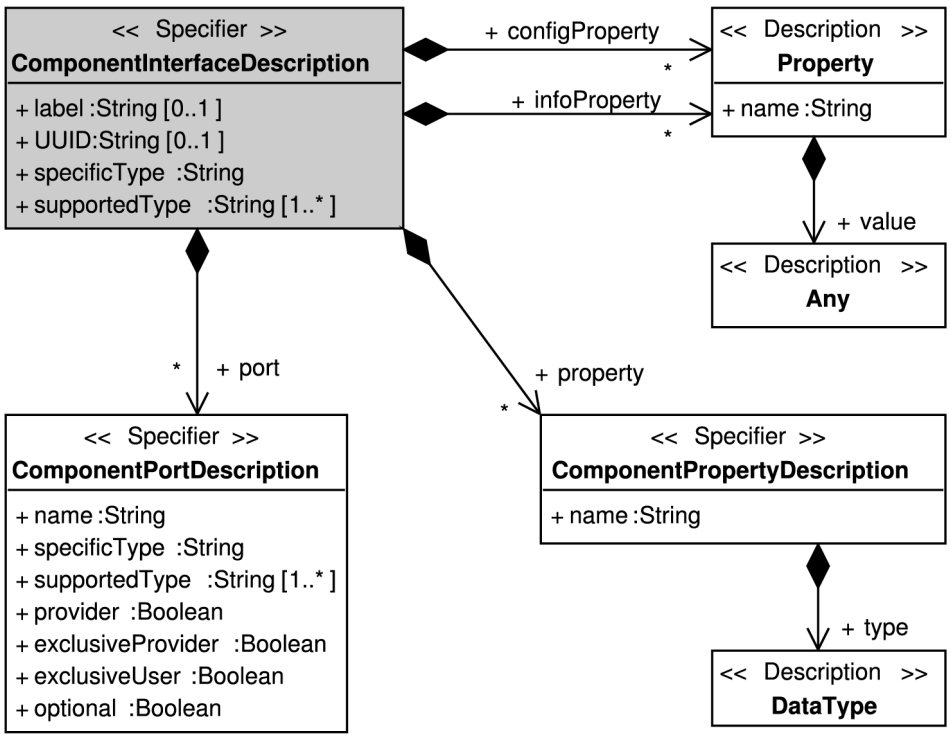


Figure 56: Detail of component interface description

6.2.2 Execution Data Model

The execution data model captures the result of combining a component data model of a particular application with a target data model of a particular domain.

The combination takes place during the planning stage of the deployment process, and the result captures the physical structure of an application in the target environment, i.e. which instance of a component implementation will run on which node. The instance of an execution data model is referred to as deployment plan, and the information it contains is used by execution management entities during preparation and launch stages of the deployment process.

Figure 58 shows a high level overview of the execution data model with additional details exposed in some of the classes. The deployment plan is analogous to the description of component assembly, and in fact contains a flattened view of the top level component assembly which represents the whole application. Of the logical structure of an application originally contained in an instance of the component data model, only the information required to create component instances and connections is retained.

The classes in the execution data model, which capture the composition of individual artifacts into component implementations, instantiation of components, and the connections among components, are similar to classes in the component data model, but not identical. The mapping between the classes in the two models is not explicitly defined and remains at discretion of the planner tool which is responsible for the transformation of an instance of the component data model into instance of execution data model.

This adds a significant amount of flexibility to the deployment process. If e.g. the component data model is extended to support other (possibly higher level) abstractions

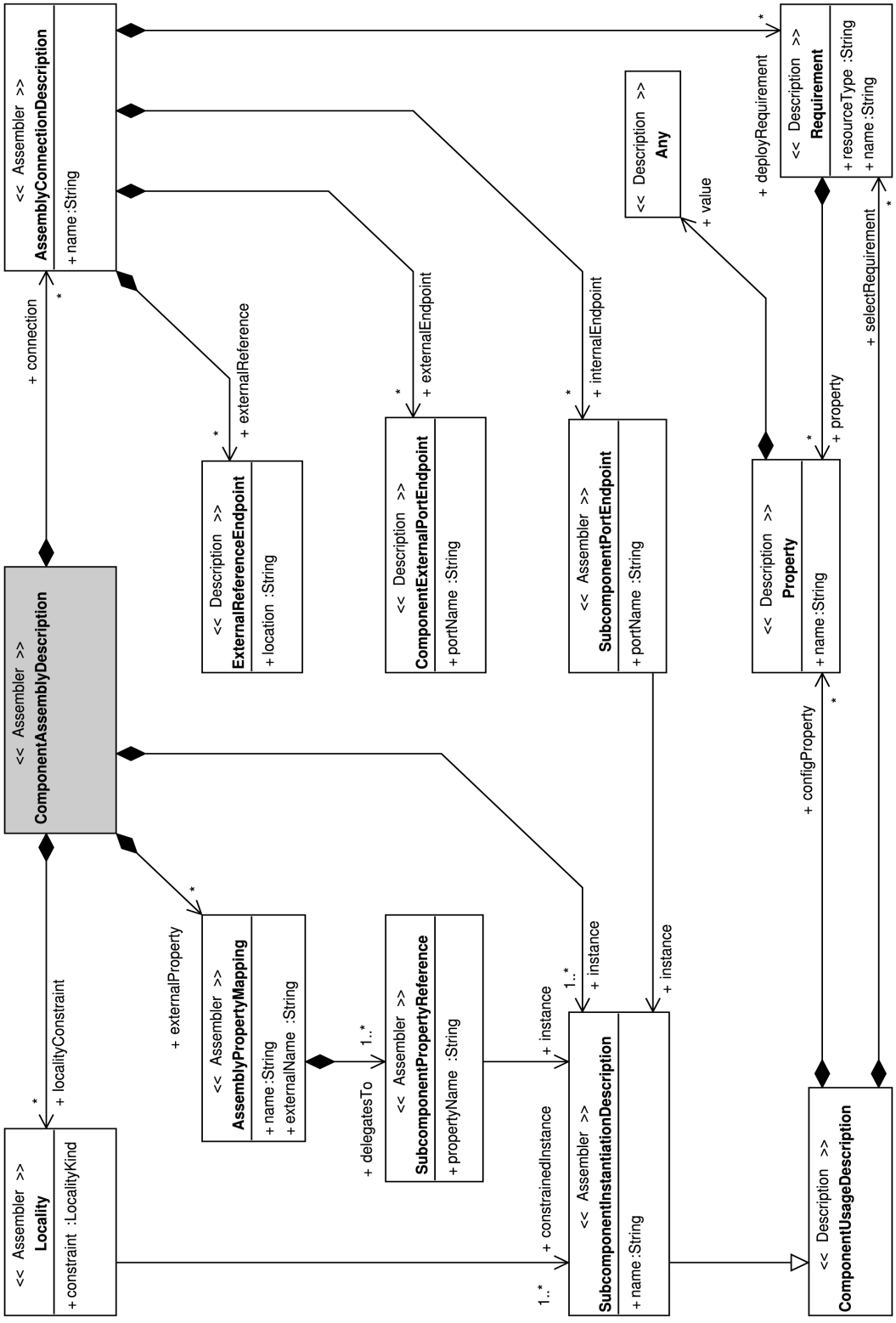


Figure 57: Detail of component assembly description

for which code can be automatically generated, the planner tool performing the transformation between the two models can also generate (or have other application do it on demand) the required code. The resulting deployment plan can be then augmented to reflect the higher level abstractions (and associated components and artifacts needed to implement them) in the physical structure of an application.

6.2.3 Deployment Process

The deployment process as defined by the OMG specification consists of five stages. Prior to deployment, an application must be developed, packaged, published by a provider, and obtained by a user. The target environment, in which an application is expected to run, consists of nodes, interconnects, and bridges. The deployment infrastructure must provide a deployment repository, in which the application package can be stored.

Installation

During the installation stage, the application package is installed into a deployment repository, which will make it accessible to the following deployment stages. The deployment repository and its locations are not related to the domain in which the application will execute. The installation stage also does not involve any copying of files to computational nodes in a domain.

Configuration

When an application has been installed into the installation repository, its functionality can be configured by the deployer. An application can be configured multiple times for different configurations, but the configuration should not concern any deployment related decisions or requirements. The configuration stage is meant solely for functional configuration of an application.

Planning

After installing and configuring an application in the deployment repository, a deployer can start planning the deployment of an application. The process of planning involves selection of computational nodes available to an application, assignment of component instances to computational nodes, allocation of node resources required for execution, choosing from multiple implementations of component instances, etc. The planning does not have any immediate effect on the domain.

The result of planning is a deployment plan, which is specific to the target domain and the application being deployed. The plan is produced by transforming the information from the component and target data models into execution data model.

Preparation

Unlike the planning stage, the preparation stage of the deployment process finally involves performing work in the target environment in order to prepare the domain for execution of an application. If an application is to be executed more than once based on a single deployment plan, the work performed during the preparation stage is reusable. The actual moving of files to computational nodes in the domain can be postponed until the launch of an application.

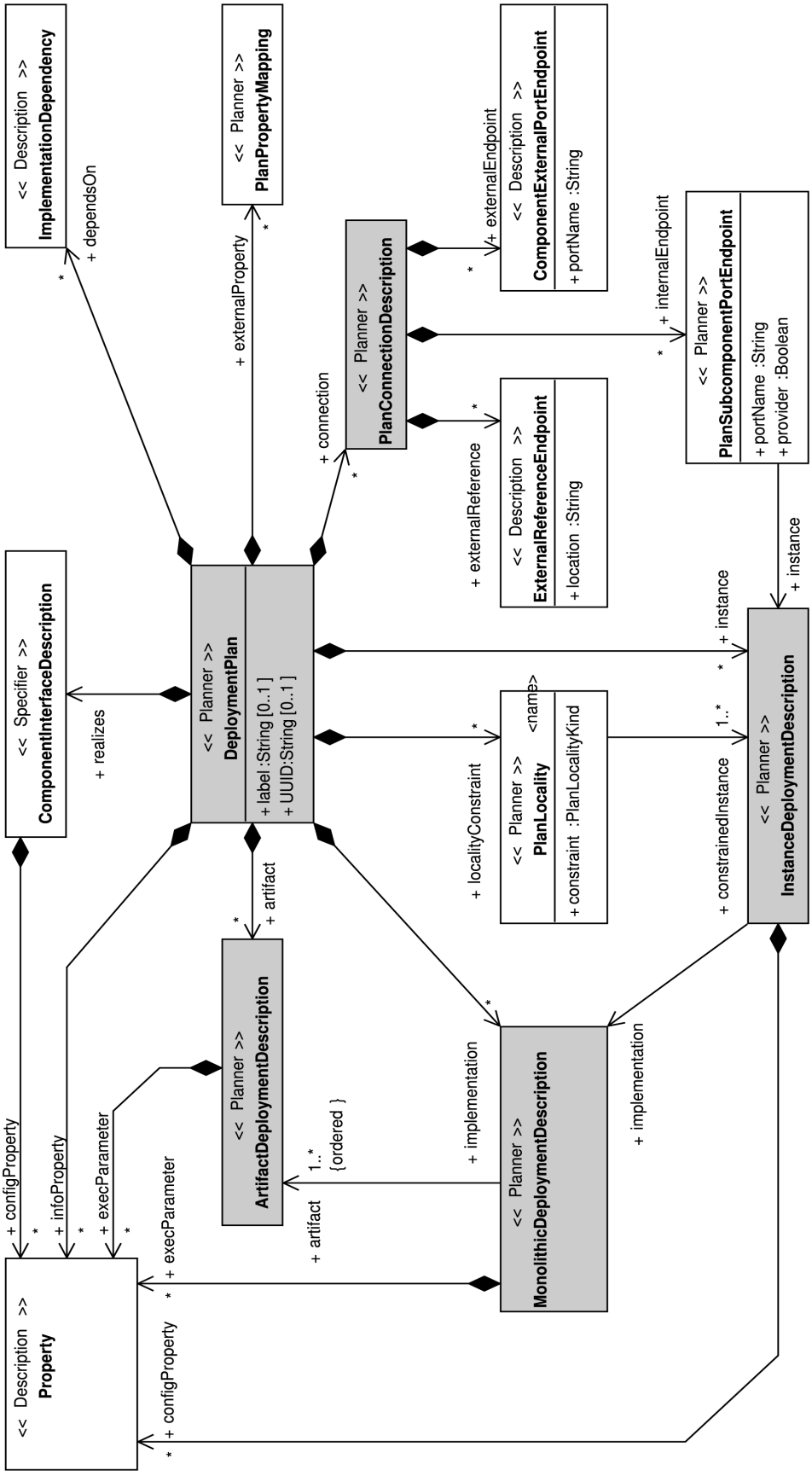


Figure 58: Overview of execution data model

Launch

An application is brought to the executing state during the launch stage of the deployment process. As decided during planning, instances of components are created and configured on selected computational nodes and the connections among the instances are established. Finally, an application is started and runs in the domain until terminated.

6.3 Integrating Connectors with Deployment

The OMG specification is very comprehensive, but also fairly complex. However, we only have to deal with parts of it to enable the use of connectors in the deployment process.

Connectors implement connections and thus mediate communication among components. The implementation of connectors is expected to be generated by a connector generator with respect to requirements on the connection a connector is supposed to implement. To capture the information necessary to generate a connector implementation, the component data model, which holds metadata concerning the logical structure of an application, needs to be extended to enable specification of connection requirements. These requirements determine connector features for every connection (communication style and non-functional properties).

Concerning the integration of connectors into the deployment process, the most suitable is the planning stage. At this stage, using a planner tool, a deployer plans the deployment of an application into a domain. The planner tool accesses application metadata as well as metadata describing the target domain through the deployment repository and lets the deployer to assign components to computational nodes in a domain. The assignment is validated by the tool to ensure that all deployment requirements of individual components have been satisfied and that there are sufficient resources on the computational nodes to allow hosting the respective components. With the exception of connection requirements, the planner tool has all information required for generation of connectors. The integration of connectors is therefore achieved by extending the planner tool to interpret connection requirements both during planning of application deployment and during generation of deployment plan.

The last aspect of connector integration is related to the process of preparing and launching an application. This part of the process is covered by the execution management model, which has not been described in this thesis. However, for now it suffices to say that execution management model consists of domain-wide and node-local execution management entities which operate on the deployment plan produced at the end of the planning stage of the deployment process. These entities cooperate to create a running instance of an application in the target domain, which consists of several node-specific parts, managed by the node-local management entities. The important point is that the primitives in a deployment plan are interpreted by the node-local entities. These entities represent the part of the deployment runtime that needs to be designed to support components from different components systems.

6.3.1 Specification of connection requirements

To generate a connector implementation, a connector generator needs to have enough information concerning the requirements on the communication a connector is expected to mediate. The requirements are expressed in form of a communication style, which determines the communication paradigm for the connection, and non-functional properties, which may impose additional requirements on the realization of the

connection. Each connection among component instances in an assembly can have different requirements.

The original platform independent component data model defined in the OMG specification requires a minor extension to enable specification of connection requirements. To accommodate that, we have added an association to the ComponentPortDescription and AssemblyConnectionDescription classes of the model named connectionRequirement, as illustrated in Figures 59 and 60, respectively.

The association is structurally equivalent to the deployRequirement association already present in the AssemblyConnectionDescription class. The connection requirements associated with the AssemblyConnectionDescription class determine the features of a connector which will implement the connection. In case of the ComponentPortDescription class, the requirements are specific to a particular component interface port.

The reason for not using the existing deployRequirement association is to have an association with clearly defined semantics in both classes, which would not be possible in case of the AssemblyConnectionDescription. Throughout the specification, deployment requirements are associated with availability of resources on computational nodes in a domain and the contents of the deployRequirement association are expected to be matched against requirement satisfiers describing available resources.

These extensions of the component data model are fairly minor and non-intrusive. We have specifically avoided using higher-level constructs such as special hierarchy of classes that would model connection requirements, because it makes the extension future proof with respect to development of tools realizing this approach. Currently, the specification of connection requirements should reflect the high-level connector specification introduced in [BP04] and applied in context of a real connector generator in [Bur06].

6.3.2 Connector aware planner tool

The planning stage of the deployment process is probably the most powerful concept of the specification. This is caused by the transformation of the component data model (logical structure) into execution data model (physical structure) which is not explicitly defined in the specification.

During the transformation, the planner (or its extensions) can generate additional code artifacts, component instances, connections among components, and resolve indirect artifact or component package references before transforming the logical structure of an application into its physical representation necessary for deployment and execution. This allows introducing higher level abstractions into the component data model without the need to modify the execution data model, as long as it provides primitives sufficient for representation of the higher level abstractions in the deployment plan.

Connectors are an example of such higher level abstraction. While the original specification expects direct connections among component interface ports, indirect connections implemented by connectors can be achieved by modifying the planner (or providing an extension) to interpret the requirements of individual connections, synthesize a connector implementation with features reflecting the requirements, and include the necessary primitives, such as additional artifacts and connections, in the resulting deployment plan which can be used to execute an application.

Besides deriving connector features for implementation of a particular connection, the semantics of connection requirements often influences the planning process. For example,

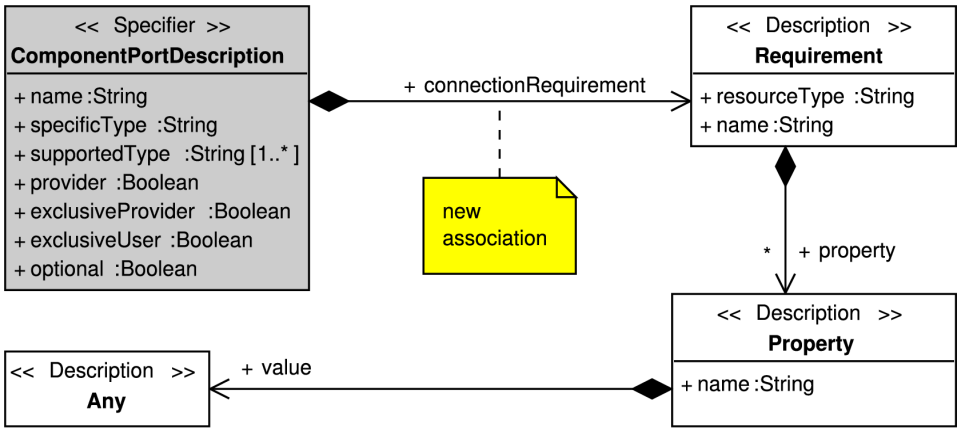


Figure 59: Modification of the ComponentPortDescription

connection requirements may specify requirements that cannot be satisfied when a pair of component instances is assigned to particular nodes.

Consequently, in addition to deployment requirements, the planner tool must also validate connection requirements specified in the extended component data model, to disallow assignment of components to computational nodes violating the requirements imposed on a particular connection. The planner tool therefore needs to be modified (or provided with an extension) to interpret the semantics of the connection requirements.

Most of the work involving generation of connectors will be performed at the end of the planning stage, when an application has been successfully planned for deployment into a domain and a deployment plan for the application is about to be generated. This is the most suitable moment for generating connectors, because all the necessary information, such as assignment of components to nodes and capabilities of the nodes, is available.

The process responsible for generating implementation of connectors attached to components from a particular component system is also responsible for generating elements of the deployment plan describing the physical representation of the connector implementation. The representation of connectors in a deployment plan must ensure correct instantiation of connectors and allow establishing connections among components at application launch.

6.3.3 Deployment runtime with connector support

The process of preparing and launching an application is realized by a domain-wide execution management entity named ExecutionManager, which in turn uses node-local execution management entity named NodeManager running on each node in the domain to realize deployment and execution of application components assigned to a particular node. The process is initiated by submitting a deployment plan to an executor tool, which reads the plan, locates ExecutionManager, and provides it with the deployment plan.

The ExecutionManager splits the deployment plan into parts corresponding to nodes hosting component instances, and disseminates the plan to NodeManager entities running on the respective nodes. A NodeManager interprets the node-specific part of a deployment plan and provides an ExecutionManager with a representation of the node-specific part of an application. The ExecutionManager then aggregates these node-specific

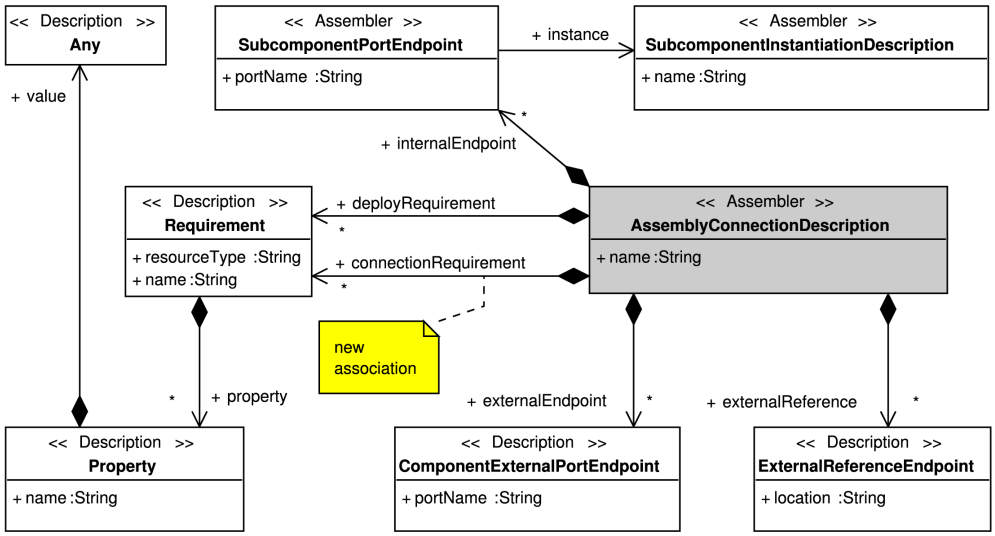


Figure 60: Modification of the AssemblyConnectionDescription

parts into a single representation of a running application in a domain. Further details concerning the process can be found in the specification.

The deployment plan is interpreted by a NodeManager, which therefore needs to support different component runtimes. Since the entities of the execution data model are not semantically equivalent to the entities of the execution data model, the semantics of the primitives contained in a deployment plan can be interpreted in different ways by different component systems. The NodeManager can support different component systems through a system of extensions, as described in [Saf07]. The rest of the deployment infrastructure above the component systems support is generic, does not interpret the semantics of deployment plan primitives and mainly serves for coordination and transport of data. In principle, the generic part of the infrastructure only processes the deployment plan on syntactical level.

All this allows the representation of a connector in a deployment plan to be tailored to a target component system runtime in which the connector is intended to be instantiated. Concerning the deployment infrastructure, this representation must only adhere to a limited semantics of deployment plan primitives to allow the management entities to coordinate the exchange of data during application preparation and launch. To illustrate the process of preparation and instantiation of connectors, consider a simple application using connectors to mediate communication between a number of client components and a single server component depicted in Figure 61.

During the planning stage of the deployment process, a deployer uses a connector-aware planner tool to plan the deployment of the application to the target domain. The planner tool validates the assignment of components to individual nodes ensuring that both deployment and connection requirements associated with components and connections among them are satisfied. When the planning is complete, the planner tools provides a connector generator [BP04, Bur06] with information necessary to synthesize a connector implementation for each connection in the application, such as the connections between the client components and the server components depicted in Figure 61. The information consists of the connection requirements specified in the component data model describing the application, assignment of components to individual nodes in a domain, and information on resources available on each of the nodes, which can capture e.g. the

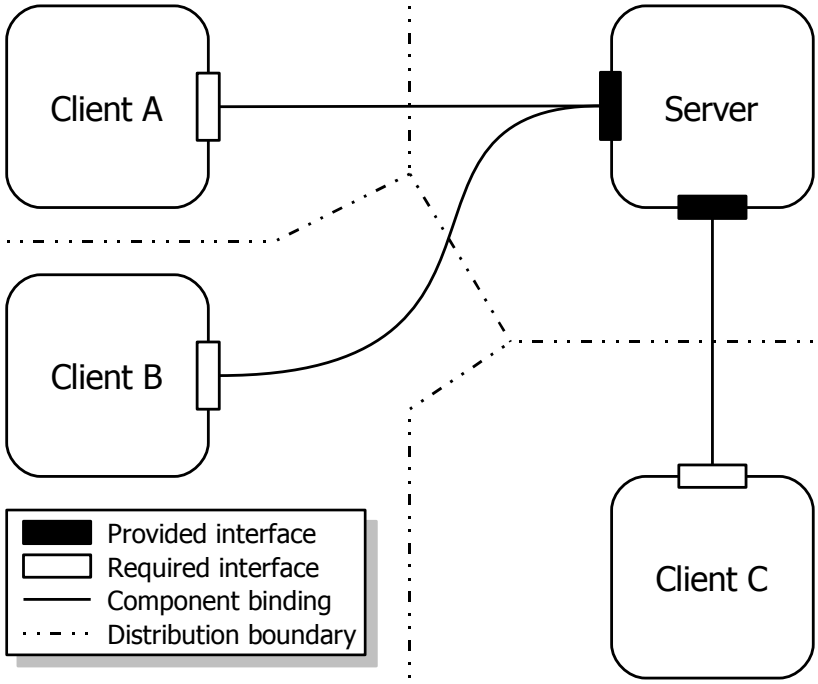


Figure 61: Logical structure of an application

availability of a particular middleware platform that can be used by connectors to achieve distribution.

Using the information provided by the planner tool, a connector generator produces code implementing the connectors, along with additional data files needed for correct instantiation of connectors at runtime, such as the connector unit instantiation template, which assigns connector units to component types and component instances, and a remote binding map, which describes the connections among connector units that together represent a connector as a distributed entity. The planner, using extensions for each of the component systems used in the application, creates a representation of a connector in the deployment plan with semantics specific to each of the component systems. The connector code, connector unit instantiation template, and remote binding map are encapsulated in implementation artifacts associated with implementation of component instances used to represent connector units. However, the semantics of component instances representing connector units differs from that of the component instances corresponding to application components described in a component data model. Deployment plan component instances corresponding to connector units merely represent entities that can be referred to and that can participate in the exchange of component references during application launch. Their role in the runtime structure of an application is determined by the implementation of a particular component system runtime.

To illustrate the process, Figure 62 shows the representation of connectors in a deployment plan created for the application originally depicted in Figure 61. During the transformation of the application component data model to the execution data model represented by deployment plan, the planner (and its component system specific extensions) transformed the original connections between application components into instances of connector units connected to their respective components. The connection between the connector units corresponds with the original connection, now implemented by a connector. The connections between connector units and their respective

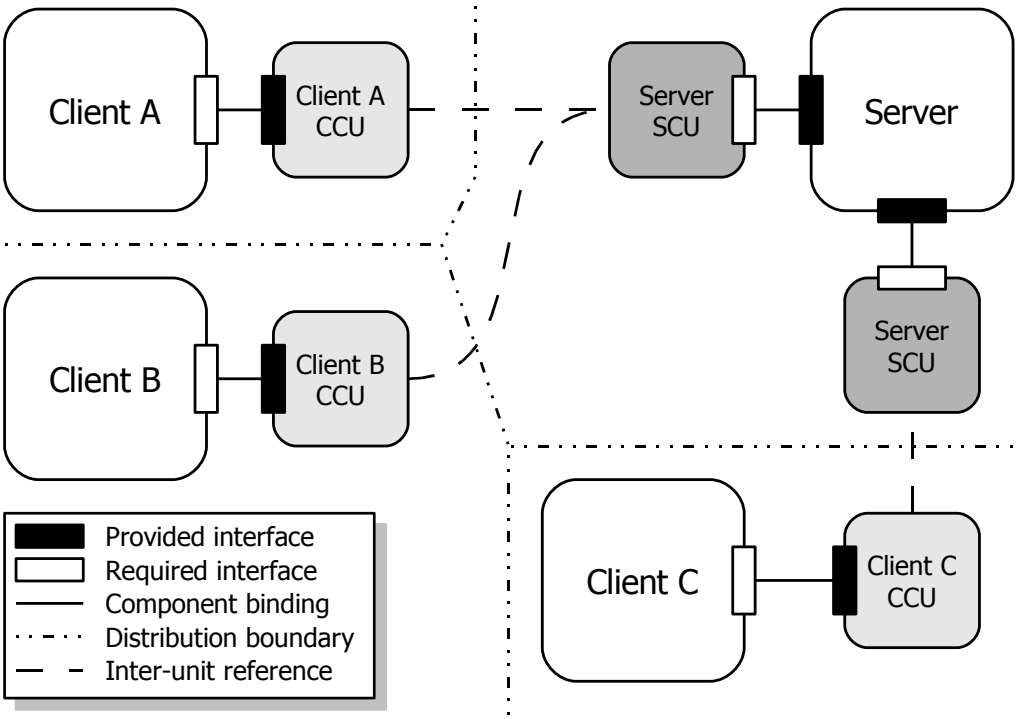


Figure 62: Physical structure of an application with connectors

components are strictly local and are used for binding connector units to component instances.

Runtime instantiation of connectors is the responsibility of each component runtime supported by a NodeManager. The generic part of a NodeManager requests instantiation of components as defined by the deployment plan, preserving the ordering. The NodeManager uses an identification of a component system corresponding to a particular component instance to determine which component system runtime should create the instance. The component system identification is part of execution parameters associated with every component instance. The extensions of the planner tool responsible for representation of a connector in their respective component system may include additional parameters required by to component runtime to correctly interpret the semantics of execution data model component instances. This approach has been applied in the implementation of execution runtime for heterogeneous applications [Saf07].

Even though the instantiation of connectors depends on a particular component system runtime, in the context of connectors and component models used in our research, the instantiation of connectors is generally based on the process associated with the platform independent model of connector runtime described in [Bur06].

6.4 Using Connectors to Bridge Component Differences

Connectors are generally useful for mediating communication among components in because they decouple the implementation of component functionality from the implementation of the communication style. Especially in distributed environment, connectors allow postponing the choice of communication middleware used to accomplish distribution because connector implementation using a particular middleware can be generated based on the capabilities of the target execution environment. Since the

execution environment becomes known at application deployment, the deployment process is a natural place for integration of connector generation.

In addition to providing implementations of component connection, the role of connectors in the deployment process of heterogeneous component-based applications is to provide a bridge for incompatibilities between components from different component systems. This is a key connector feature which provides a foundation for creating heterogeneous component-based applications.

6.4.1 Prerequisites for heterogeneous deployment

Primarily, connectors represent a concept which, being well applicable in the context of component-based software development, serves mainly as a container for technical solutions that enable bridging component system differences during assembly and deployment of heterogeneous component-based applications. Consequently, there are several prerequisites to using connectors to bridge component system differences.

Supporting connectors in component systems

First of all, using connectors to mediate communication between components from different component systems requires the respective component systems to support connectors. Even though the majority of existing component systems does not provide connector support, it should be possible to add connector environment to most component systems, because the basic concept of connectors is rather general. However, in some cases it may be too complex to provide practical benefits.

This has been demonstrated in case of the SOFA and Fractal component systems in [Bur06]. Unlike SOFA, which supports the concept of connectors natively, the Fractal component model has no connector support. The connector environment for both component systems was based on a platform independent model of a connector environment, which has been then specialized to a particular component system.

The situation is more difficult in case of the EJB component system which, in comparison to SOFA and Fractal, is based on different concept of components. This is evident mainly from the fact that the EJB component model supports several kinds of components with different life cycle management and that there is no concept of explicit application architecture description. We have shown that in principle, connectors can be supported at least for stateful session beans [BB04], but the work has remained in a proof-of-the-concept stage, because EJB is not a primary target of our research.

However, adding connector support to the EJB component system has revealed the most problematic issue related to connector support in component systems, which is conceptual incompatibility. Such incompatibility is difficult to address through technical means and changing the fundamental concepts in a commonly used component system, and thus obsoleting the entire industry behind the component system, would not make any sense.

Establishing connections among components

Having connector support in respective component systems does not automatically enable connecting components from different component systems. To enable communication among components using a connector, the connector units attached the components must be able to exchange references and establish connections. In some situations, the required references may be unavailable.

This is an issue related to compatibility between the life cycle management of individual component systems and the life cycle management of the deployment runtime. In case of a homogeneous deployment runtime the life cycle management is dictated by the (only) supported component system. In case of heterogeneous deployment runtime, the life cycle management is dictated by the runtime and may be thus incompatible with certain component systems.

The launch stage of the deployment process defined in the OMG specification provides a two-phase procedure for exchanging initial references among components, which should allow the underlying component system runtimes to ensure proper instantiation of components and connector units as well as bindings among them.

Nevertheless, even at this stage, conceptual differences that are difficult to solve may be encountered. For example, in SOFA and Fractal, components are instantiated at application launch to create an initial architecture of an application and there is no problem with exchanging references during the launch stage of the deployment process. This concept is however completely foreign to EJB. Components in EJB applications are instantiated either by other components (entity beans) or by the application server (session beans) in response to a connection from a client, which in turn is a concept totally foreign to SOFA and Fractal. Consequently, it is difficult to obtain initial references to EJB components with the semantics corresponding to references in SOFA or Fractal.

When deploying and launching heterogeneous component-based applications using a generic deployment runtime, these issues limit the spectrum of component systems that may be supported by the runtime.

Understanding each other in communication

The final step in enabling connector-based communication among components from different component systems is to ensure that all components participating in the communication are able to interpret the same data in the same way. This is not a problem of using a particular middleware to connect distributed components, because connectors can employ almost arbitrary middleware to implement a connection according to a particular communication style.

This problem is related to compatibility of types exposed in the interfaces through which the components communicate. When connecting components from different component systems, their interfaces may be specified using types that only exist in a particular component system. This includes both types native to a programming language and types specific to a particular runtime environment.

Distributed middleware platforms such as CORBA have approached this issue by requiring all interfaces for distributed communication to be defined using an interface definition language, which provided a unified type system for all runtime platforms, independent of programming language or runtime library. This type system was then mapped to all supported programming languages. However, this approach only allows exchanging a very limited set of types in distributed communication, to allow mapping to all supported languages. This restriction can be lifted by using language specific middleware such as SUN RMI. On one hand, this allows using almost all types (they need to be serializable) supported by the runtime, but on the other hand the distributed communication is restricted only to systems implemented in the same language.

Component systems are mostly typically intended for assembly of homogeneous applications; in some cases they are intended for a single programming language, which allows them to use the type system of the underlying language and runtime. This approach

is perfectly legal in the sense that a particular component system may be designed for a particular problem or application domain, which in turn allows tailoring the design of a component system to the commonly used practices or programming languages used in that particular domain. However, it is a problem when attempting to connect components from such component systems.

The component interfaces have been already defined, so requiring them to be redefined using some form of an interface definition language, especially when that language imposes restrictions on types that can appear an interface, is not feasible. Instead, when connecting component from different component systems, a mapping between the type systems of components participating in the communication must be created, one for each pair of different type systems. Compared to the IDL-based approach, which only requires O(N) mappings to be defined, this approach requires $O(N^2)$ mappings, with N being the number of different type systems.

When connecting components using connectors, a connector generator must be able to map types between the type systems of the communicating components. A particular approach implemented in a connector generator for the connector model used throughout this thesis is described in [Bur06]. The generator uses an abstract type model into which different type systems may be plugged. Mappings between type systems are realized by functors applied to type specifications which must be implemented by type system plugins.

In our particular case, mapping types between the SOFA, and Fractal component systems is relatively easy, because they are both Java-based and conceptually similar. The EJB component system is conceptually different from SOFA and Fractal, which may lead to problems with mapping component system specific types. However, being Java-based like SOFA and Fractal, these problems may be easier to resolve.

In general, as long as component interfaces only use “non-problematic” types, such as those that would be supported by an IDL, the mappings can be created automatically. However, for certain complex types, special adaptation may be required, which is easier to perform for semantically equivalent types. In addition to middleware or instrumentation code, connectors may also contain adaptation code, but this code cannot be always generated automatically.

Type system adaptation is a difficult problem that probably does not have a general solution. However, a solution can be found for many practical applications, as evidenced by the variety of existing middleware bridges.

6.4.2 Realistic heterogeneous deployment

Due to various issues, satisfying the prerequisites for using connectors to bridge component system differences may seem impossible. Aside from connector support, the main issues in connecting components from different component systems are related to life cycle and type system compatibility.

The problem of incompatible type systems is mainly a technical issue in communication and in most cases it can be addressed by technical means (automatic or manual adaptation), depending on the situation. On the other hand, incompatible life cycle is mainly a conceptual issue, which is more difficult to address by technical means. In this context, an interesting work focused on automatic identification of compositional conflicts between component systems is presented in [LBS05]. This is important because when

trying to bridge differences between component systems, these differences have to be first identified.

In general, we are aware of the complexity of both issues, however in the context of our research on component-based software development, we are not aiming at finding a universal solution, which may not even exist. We work mainly with the SOFA and Fractal component systems which are conceptually similar and based on the same programming language; therefore we are typically faced with problems that can be solved through technical means. We may become interested in applying the approach to a different component system, but due to the nature of our research, such component system would be also conceptually similar to the systems we already work with, even though it may be based on different programming language.

From the point of view of software development practice, we also do not expect heterogeneous component-based applications to establish a dense network of connections among components from different component systems. Rather, we expect that different component systems may be used to implement application subsystems and that the communication between those subsystems will be realized by a limited set of components. In such cases, the communication interfaces may need to only use a restricted type system similar to that of distributed middleware platforms, which can be mapped between all participants in a communication.

6.5 Integrating Performance Instrumentation with Deployment

Due to their compositional nature, connectors can be used for generic instrumentation of component-based applications. Consequently, connector-based performance instrumentation as described in Chapter 5 can be generally applied to component applications based on different component systems. With connectors bridging incompatibilities between different component systems, the instrumentation technique becomes applicable even to heterogeneous component-based applications.

Because the performance instrumentation code is produced by a connector generator along with code implementing the communication style of a particular connection, the integration of performance instrumentation into application life cycle is basically subsumed by integration of connectors and connector generation into deployment process, as described in previous sections of this chapter.

The only aspect of performance instrumentation process that is not inherently addressed by the integration of connectors into deployment is the decision to instrument an application. This decision needs to be reflected in connection requirements associated with connections among components to ensure that a connector generator will produce connector code with performance instrumentation elements.

In the context of the deployment process defined in the OMG specification, this can be addressed either by creating a simple configuration tool which will add the necessary connection requirements to the component data model of an application, or it can be addressed by adding an option to instrument an application to the planner tool so that it will automatically include connection requirements related to performance instrumentation in connection requirements passed to the connector generated at the end of planning. In either case, this is basically an issue of implementation, which does not prevent transparent integration performance instrumentation in the deployment process.

Chapter 7

Evaluation and Related Work

The following sections provide an evaluation of the proposed conceptual framework with respect to the objectives and requirements elaborated in the initial chapters of this thesis, and a comparison of the approaches we have adopted to address the individual subgoals of this thesis with approaches found in related works with similar goals.

7.1 Evaluation of the Solution

The key aspects of the problem, related to access and storage of performance data, definition of performance events driving the measurement and, in context of component-based applications, also issues related to application deployment, determine the decomposition of the main goal of this work into subgoals, with specific requirements and objectives.

In turn, the decomposition determines the role of the solutions to individual subgoals in the overall solution. Therefore to achieve to goal of this thesis, we provide the following solutions to the subgoals of the main goal:

- a design of a generic measurement infrastructure responsible for handling collection and storage of performance data and capable of associating performance data from diverse sources with performance events,
- a design of connector-based application instrumentation technique which provides performance events driving the collection of performance data, and
- heterogeneous application deployment, which serves as an integration platform for the instrumentation and the approach as a whole.

Even though they are often presented and evaluated separately, these solutions depend on each other, because decisions made to address one issue needed to be considered in the context of solutions to other issues. Together they provide an approach to obtaining design-level performance data from heterogeneous component-based applications which satisfies the general requirements on being generic, non-intrusive, and transparent. We therefore consider the main goal of this thesis accomplished.

To support this claim, we evaluate the presented solutions to individual subgoals of the thesis in context of their specific requirements and objectives.

7.1.1 Measurement infrastructure

The design of the measurement infrastructure proposed in Chapter 4 provides a unified interface for accessing different performance data sources, and covers the architecture and interfaces of individual subsystems responsible for processing different types of performance events, storage of performance data associated with events, and runtime

control over association of performance data with performance events as well as control over the type of events that should be emitted by various performance event sources.

The concept of event sources allows using the measurement infrastructure both from standalone applications and from performance instrumentation code, without regard to the underlying component runtime. The management part of the infrastructure is designed to allow configuration of all aspects of performance data collection at runtime, which makes it suitable for use with long running systems in production environment and also simplifies experiments related to application of measurement-based performance analysis techniques. The measurement infrastructure thus satisfies the requirements for a generic approach both from the main goal and from the first subgoal of the thesis.

The design of the measurement infrastructure strictly separates the provided and required interfaces for different roles assumed by subsystems of the infrastructure. This alone makes the infrastructure flexible with respect to different implementations of its subsystems. At runtime, the infrastructure is mainly extensible with respect to accessing different performance data sources. This is realized by the Performance Data subsystem, which allows registering additional data source modules to provide access to additional data sources. Even though it has not been elaborated in detail, the infrastructure has been designed with different data delivery methods in mind. For increased flexibility, an interface allowing dynamic registration of data delivery methods may be provided to users of the infrastructure. However, mainly due to its interface-based design and extensibility of the Performance Data subsystem, we consider the infrastructure to satisfy the requirement for an extensible design from the first subgoal of the thesis.

The low-level parts of the Performance Data subsystem responsible for accessing various data sources are inherently platform specific. The high-level parts of the subsystem, including the rest of the measurement infrastructure, can be implemented in the target programming language, using only standardized network or file system interfaces through the usual portability glue. Other than that, the concepts used in the design of the measurement infrastructure do not depend on the specifics of a particular platform. Even though an implementation of the infrastructure is not yet available, there is nothing that would make it inherently non-portable, therefore we consider the requirement for portability from the first subgoal of the thesis satisfied.

Access to different data sources

The first group of requirements concerning the measurement infrastructure is related to the part responsible for integration and access to various data sources.

Time source needed for attaching essential timing information to performance events can be accessed through an abstract interface, which allows generic access to an implementation providing a high-resolution time source available on a given platform, with the option to fall back to a generally available but low-resolution time source.

The concept of performance data source modules that can be registered with the infrastructure provides the data access layer with extensibility regarding different performance data sources. The performance data modules provide a generic management interface which can be used to query and access the available performance data.

The performance data are represented using a unified naming scheme, with each data source module responsible for its own name space of sensor groups and associated sensors. The layout of the group name space is in the discretion of the module, but the infrastructure provides a generic way to query the available performance sensors.

The concept of a measurement context decouples selection of sensors that should be accesses from sampling and decoding of performance data obtained from underlying data sources. This allows each data source module to prepare an optimal strategy for efficient access to its performance data sources when requested to obtain the values of selected sensors. Moreover, the concept of a measurement context is essential in that it enables the measurement infrastructure to associate arbitrary performance data with performance events. The association needs to be created only once for all subsequent occurrences of a particular performance event.

The sensor naming scheme along with an interface that allows data source modules to access other modules enables creating special data source modules that provide sensor name aliases. This can be used to create a data source module providing sensors with platform independent names which will be internally mapped to corresponding sensors provided by platform specific data source modules. This mapping is only resolved at measurement context creation time, therefore it does not incur any overhead during sampling and decoding of performance data, which is performed by the data source modules responsible for particular sensors.

The above evaluation shows that the performance data access layer satisfies the specific requirements on this part of the measurement infrastructure and thus meets the objective to design a performance data access layer providing the measurement infrastructure with a unified interface for accessing performance data from diverse data sources.

Runtime configuration, management, and control

The second group of requirements on the measurement infrastructure is related to the remaining subsystems responsible for collection, processing, and storage of performance data as well as runtime configuration and management.

The management part of the infrastructure provides an interface which allows querying and enumerating the available types of performance events and performance data that can be associated with the events. The information related to performance events is obtained through the instrumentation layer, while the information about available performance data is obtained from the Performance Data subsystem of the infrastructure. The management interface then allows configuring what kind of performance data should be collected in response to a particular event occurring at a particular location.

The measurement infrastructure allows storing data in memory and supports several basic strategies for handling situations when the storage capacity is exhausted. The policies support allocating additional storage, ignoring any new data, and reusing the storage by rewriting older data records. The storage capacity and policy can be configured at runtime for each performance event. To minimize the influence on the system under test, all memory required for the storage is intended to be preallocated prior to infrastructure operation.

Delivery of performance data to consumers is not specifically addressed, because it opens up a whole new area of research with additional issues that need to be considered. Because data delivery methods are not particularly related to the problem of obtaining performance data from heterogeneous component-based applications, we consider them beyond the scope of this thesis. Since data delivery depends neither on the instrumentation nor on the application deployment, arbitrary delivery methods can be implemented on top of the measurement infrastructure and are thus considered to be an extension. However, an implementation of the infrastructure should be able to write the performance data it has collected at least into a set of files.

The above evaluation shows that the both the processing and management parts of the measurement infrastructure satisfy requirements on runtime configuration, management, and control and thus meet the objectives of the first subgoal of the thesis.

7.1.2 Performance instrumentation

The connector-based instrumentation proposed in Chapter 5 provides a generic technique for instrumentation of component-based applications. In general, connectors are used to mediate communication among components, but can also contain code to implement functionality which is orthogonal to their primary function, but is necessary for the connection represented by the connector to possess certain non-functional properties.

Connectors are attached to the boundaries of components represented by component interfaces and allow middleware independent interception of business method invocations. As such they provide a unifying concept and location for definition of design-level performance events associated with collection of performance data.

From the architectural point of view, connectors are elements of composition and thus compatible with the inversion of control principle typically employed in component-based software engineering. Consequently, connectors are generally applicable to different component models.

Connector implementation code is decoupled from the implementation code of application components and is expected to be automatically generated. The instrumentation code can be generated as a part of the connector implementation code without the need for application source code. The instrumentation is thus completely orthogonal to the original application, which makes the instrumentation process non-intrusive and transparent for application developers.

The technique depends on support for compositional connectors a particular component system and on support for that component system in connector generator.

- Connector generator is a sophisticated code generator designed for extensibility with respect to target implementation languages and platforms. Extending a connector generator to generate connectors for a new component system is easier than adding support for a new target language, because supporting a new language requires supporting its type system.
- Adding support for connectors to an existing component system may be more difficult, but the concept of connectors is high-level and does not depend on specifics of any particular platform. With certain drawbacks, the concept can be simulated using components instead of connectors without substantially modifying the underlying component system.

We believe that connectors are essential for supporting assembly and deployment of heterogeneous component-based applications, because in addition to liberating application components from the specifics of various middleware platforms, they can provide adaptation between specifics of different component models. Therefore, we do not consider the need for connector support in a particular component system to be an obstacle for generic connector-based application instrumentation.

Since the generation of connectors is integrated into the planning phase of application deployment, the instrumentation process itself is completely transparent for the person performing the deployment.

The element-based connector model along with an algorithm for connector reconfiguration provide control over the runtime overhead of the instrumentation code in connectors. The instrumentation code in connectors can be completely excluded from execution, yet the approach remains generic and does not depend on the specifics of any middleware platform or a component runtime (other than connector support).

The above evaluation corresponds to the requirements on performance instrumentation technique (see Chapter 3). Connector-based instrumentation satisfies all the requirements and meets the objective to devise a compositional instrumentation technique for defining design-level performance events in context of different component technologies. The instrumentation code in form of connector elements along with the mechanism for connector reconfiguration meets the objective to design an instrumentation layer that provides control over the execution overhead imposed by the instrumentation.

Through requirement subsumption, connector-based instrumentation also satisfies the general requirements for a generic, non-intrusive and transparent approach both in the main and in the second subgoal of the thesis.

7.1.3 Heterogeneous deployment

Heterogeneous deployment proposed in Chapter 6 extends the deployment process for distributed component-based applications defined by OMG [OMG06c] to support heterogeneous component-based applications. The extension is based on integrating support for connectors into the deployment process.

We have analyzed the original specification and proposed connector integration on the level of component data model, a platform independent model capturing the logical architecture of an application. The component data model is further transformed into execution data model, again a platform independent model capturing the physical architecture of an application. This transformation is a result of deployment planning, a phase which serves to overlay the logical architecture of an application onto execution nodes in a deployment domain. We have proposed an extension of the planning phase to include generation of connectors, because the information needed for automatic generation of connector implementations becomes available at this stage.

As a result, the extended deployment process provides a universal framework for handling heterogeneous component-based applications. This meets the objective to propose a method for enabling deployment of heterogeneous component-based applications in context of the deployment process defined by OMG and thus satisfies the general requirement for a generic approach to different component systems in the main goal of the thesis.

Within this framework, the connector-based application instrumentation is fully transparent for the person performing application deployment, because the instrumentation code is generated along with the code implementing a connector. This meets the objective to integrate performance instrumentation into application deployment process without exposing a person performing the deployment to technical details of the instrumentation and thus satisfies the general requirement for transparent performance data collection process in the main goal of the thesis.

Other than the two objectives, there were no specific requirements, because the specification of the deployment process was defined by OMG.

7.2 Comparison with Related Approaches

The organization of this section has turned out to be a nontrivial task, mainly because of the many angles that need to be considered when evaluating this work in context of other works. Even though the work presented in this thesis touches three different research areas, i.e. measurement, instrumentation, and deployment, its main contribution arises from the combination of solutions in these areas to support the proposed generic approach to collection of performance data for component applications.

The purpose of this evaluation is not a comparison with narrowly focused works in each particular area. Instead, we are mainly interested in related works that, similar to the work presented here, attempt to build on results and concepts from different areas to achieve more complex goals. For such related works, topic-centric categorization would be extremely unsuitable, because the evaluation in context of a particular related work would be scattered across multiple sections.

To provide a meaningful comparison, we only deal with selected related works that are most relevant to the topic of this thesis. The selection has been already discussed in Chapter 2, which also provides a detailed overview for each of the selected works, including technical details to prepare ground for technical comparison.

7.2.1 Component-based application-level approaches

Approaches targeted at component-based applications represent the most relevant related works. They are also very rare. We believe that the COMPAS framework and the TAU CCA Tools are the only relevant works in this category. Their requirements, especially those regarding non-intrusiveness, transparency, and collection of performance data at the design level, are similar to requirements presented in this work. However, our approach has additional requirements concerning the collection of performance data and support for heterogeneous component applications.

COMPAS Framework

The COMPAS framework [Mos04] combines performance measurement infrastructure with monitoring and diagnosis of performance anomalies (see Section 2.1 for an overview). Even though in COMPAS the collection of performance data is only a means to achieve adaptive runtime monitoring, in this work it is the primary goal. Therefore we will only focus on the COMPAS performance measurement infrastructure.

Even though many of the concepts present in COMPAS could be applied in a broader scope, COMPAS specifically targets the J2EE platform, with EJB as the component technology and Java as the execution environment. While in general the techniques used in COMPAS are not suitable to achieve the goals of this work, they are perfectly suitable in the environment, or a class of environments, targeted by COMPAS. Most differences between COMPAS and our approach are due to our focus on heterogeneous component applications, which requires generic support for different component technologies and execution environments.

Performance data collection

Concerning performance data, COMPAS only supports collection of timing information associated with design-level performance events generated by its instrumentation layer. In contrast to our approach, COMPAS does not address accessing different performance data sources, with the exception of precise time source.

Precise time sources are often platform specific, and since COMPAS targets EJB applications written in Java, it has to use native code libraries to access platform specific time sources. Because this is not always appropriate, COMPAS uses an abstraction which allows using the most precise time source available on the platform. If a particular platform is unsupported or using native code libraries is not appropriate, the implementation falls back to Java-based time sources.

The abstractions provided by the performance data access layer used by the measurement infrastructure presented in Chapter 4 support similar approach to accessing time sources and also provide a unified interface for accessing additional performance data sources. A fall back mechanism for other types of performance data than time could be implemented within a generic performance data provider, but is not specifically addressed in this thesis. The performance data access layer could be employed within COMPAS to provide additional types of performance data if necessary.

Concerning the actual collection of performance data, the design-level performance events originating in COMPAS Probes are either buffered or immediately sent to a Probe Dispatcher, depending on the operational mode of a particular probe. The Probe Dispatcher processes the events and only sends higher-level events to the Monitoring Dispatcher as JMX notifications. The higher-level events are based on the analysis of design-level events emitted by the Probes and are used to deliver performance alerts and aggregate information.

In contrast, the goal of our approach is to collect performance data with minimum influence on the system under test and let the consumers process the data. However, transporting the data to consumers may result in undesired interference which may influence the system, e.g. when transmitting the data over a network. What level of interference is acceptable for a particular application needs to be decided by the consumer who is responsible for choosing an appropriate data delivery method. Since finding the right set of delivery methods appropriate for most general use is beyond the scope of this work, we do not specifically address data delivery in our approach and leave this issue open for future work. However, to be actually useful, the infrastructure needs to be able to store the data in preallocated buffers, on top of which an appropriate delivery method can be implemented.

Application instrumentation

When instrumenting an application, COMPAS first obtains metadata describing the application structure and determines the names and types of components and the interfaces they implement. Using this information, COMPAS generates wrappers for each of the target application components and updates the application metadata to reference the wrappers instead of the original components. The wrappers correspond to Probes and provide the monitoring runtime with design-level performance events which correspond to method invocations on component's business and life cycle interfaces.

The wrapper components implement the same interfaces as the target application components, but their business and life cycle methods contain instrumentation code which emits performance events associated with context information. Since the EJB components are represented by a single class, COMPAS uses language-based class inheritance for implicit access to the original implementation of the target component business and life cycle methods. By using wrappers which inherit from the target application components, the architecture of the instrumented application is essentially identical to that of the target application, which avoids having to deal with placement of communication middleware.

Our approach uses connector-based instrumentation in which the instrumentation code is encapsulated in a connector element. The functionality of the instrumentation code is similar, except due to the compositional nature of connector elements, explicit delegation has to be used. The connector based approach is better suited for heterogeneous component-based applications, because it does not make any assumptions on the runtime representation of a component. The use of explicit delegation imposes minimal requirements on the target implementation language, e.g. it does not require support for inheritance.

To reduce impact of the instrumentation on a running system, COMPAS Probes can operate either in passive or active monitoring mode. COMPAS can also use JFluid [Dmi03, Dmi04] to add or remove instrumentation from a particular component at runtime. This approach is based on byte-code manipulation and has to be supported by the application server.

Our approach to managing the impact of performance data collection on a running system also works on two levels. First, the measurement infrastructure can be configured to collect selected performance data only at specific locations, with the rest of the instrumentation being passive. Second, using the connector reconfiguration mechanism, the instrumentation code in connectors can be excluded and included in the call paths of business methods at runtime, providing the same effect as byte-code manipulation in case of COMPAS.

Even though it requires a particular component runtime to support connectors, our approach is more suitable for heterogeneous component-based applications because the concept of connectors is generic and the connector reconfiguration mechanism only exploits the compositional nature of connectors, not a specific feature of an execution environment.

Deployment integration

COMPAS does not define application deployment, but has to take into account the J2EE platform deployment process. This process has as an input an Enterprise Application Archive (EAR) which contains all application artifacts, including deployment descriptors containing fragments of application metadata. The output of this process is an application deployed in an application server which is ready to be launched.

To examine the application architecture, COMPAS needs to access the deployment descriptors contained in the application archive. This is done either prior to deployment by directly accessing the application EAR files, or after deployment by using server side introspection [SUN06a]. The information obtained from analysis of the deployment descriptors combined with Java introspection is then used to generate the instrumentation layer.

Consequently, COMPAS is not really integrated into the application deployment process, but the process of application instrumentation is still sufficiently transparent because the deployer works with familiar concepts such as EAR files. However, even though conceptually portable to similar frameworks, this approach is still coupled with the properties of a particular component technology and execution environment, which in this particular case needs to support introspection.

In contrast, the approach proposed in this work is based on a generic deployment process for component-based applications, extended to support heterogeneous component-based applications (see Chapter 6). Within the deployment process, there is a unified model of application metadata, which allows tools such as deployment planner or connector generator to access and manipulate the metadata in a generic way. In conjunction with a

connector-based instrumentation (see Chapter 5), application instrumentation can be transparently integrated with the deployment process, because the instrumentation is a part of connector generation which occurs during transformation of application metadata into deployment plan.

TAU CCA Tools

The TAU CCA Tools [MS+05, PRL05] is a package which enables using the TAU performance system [SM06, PRL07a] for measurement-based performance analysis of component-based applications written using the CCA platform (see Section 2.1 for an overview). In contrast to e.g. COMPAS, the goal of TAU CCA Tools is more aligned with the goal of this thesis, which is collecting performance data for component-based applications.

In principle, TAU CCA Tools provide performance instrumentation for CCA components which uses TAU for performance measurement. Because of the multi-platform nature of CCA, TAU CCA Tools attempt to leverage advantages of component-based development in addition to platform-specific instrumentation techniques. The differences between TAU CCA Tools and the approach presented in this thesis are mainly due to our focus on generic support for various component technologies and the different nature of parallel and distributed computing environments.

Performance data collection

The collection of performance data in TAU CCA Tools is handled by the *TauMeasurement* [SM+03] and *Mastermind* [RT+04] components used by the performance instrumentation code. Together, these components provide services similar to that of the measurement infrastructure proposed in this thesis.

The *Mastermind* component handles the collection and storage of performance data as well as management of the instrumentation layer, which corresponds to the functionality of the Infrastructure Management, and Data Storage subsystems of the measurement infrastructure, while the *TauMeasurement* component defines measurement semantics and provides performance data, which corresponds to the functionality of the Event Sources, Event Processing, and Performance Data subsystems of the measurement infrastructure.

The Performance Data subsystem only provides access to performance data from different sources and the Event Processing and Data Storage subsystems are responsible for sampling and storing the performance data in response to performance events. The concept of measurement context introduced in the Performance Data subsystem allows selecting what performance data to collect for each measurement point. In contrast, the measurement port of the *TauMeasurement* component, through its Timer and Event interfaces, only provides predefined measurement semantics, and the Control interface, even though it allows disabling and enabling specific timers, does not provide any control over the kind of performance data collected by the TAU framework in response to the measurement events.

Encapsulating the performance data collection functionality into components provides an advantage in that it can be easily reused, even in CCA applications composed of components written in different programming languages. In contrast, we expect that most of the measurement infrastructure will be implemented natively for each target language, which represents an increased implementation burden. However, this is to allow e.g. a purely native implementation (if desired) in virtual machine based environments.

Since the collection of performance data is technically handled by the TAU performance system, we will only focus on the instrumentation and deployment aspects of the TAU CCA Tools approach and provide a comparison with TAU separately.

Application instrumentation

As mentioned in the overview, TAU CCA Tools can provide performance data related both to black-box (design-level) and white-box (implementation-level) view of components. Obtaining implementation-level performance data from heterogeneous component-based applications would require using different platform-specific instrumentation techniques, which conflicts with the requirement on generic approach with respect to different component models. Hence our approach is only focused on obtaining design-level performance data.

The instrumentation technique based on port wrappers is similar to our connector based approach in that instrumentation code is associated with each component port, either required or provided. However, given that CCA supports assembling only a single-address space applications, this only provides the ability to add caller context to performance measurements related to provided services. Since the approach is based on instrumenting code representing a component interface, it is only suitable of environments such as CCA, because it does not have to consider middleware code which typically represents the component boundaries in a distributed environment.

In our connector-based approach, we have control over the placement of the instrumentation code with respect to middleware code and in addition to caller context, associating instrumentation code with both provided and required ports can provide information on communication overhead related to middleware and network latencies. And finally, the connector-based approach uses a generic concept leveraging the compositional nature of component-based applications and does not depend on platform specific instrumentation techniques.

The instrumentation technique based on proxy components is similar to the wrapper approach used in COMPAS. However, unlike the wrapper components in COMPAS, the proxy components have to be explicitly included in the description of application architecture to impersonate the original components. This approach does not allow extracting caller context from performance data, because the proxy components are always associated with service provider and not with (potentially multiple) service users. Again, this approach would be problematic in a distributed environment, because of issues with placement of middleware code. Any middleware code would need to be explicitly linked with a proxy component instead of the original component, making the original component unusable without its proxy. Consequently, while this approach is perfectly applicable in the context of the CCA platform, it is not generally applicable to other component platforms.

In connector-based component systems, middleware is encapsulated in connectors, always leaving the components free of middleware code. Being first class entities, connectors are part of the application architecture; placing instrumentation in connectors thus does not change the application architecture. In addition, the compositional character of the particular connector model used in our approach provides control over placement of both middleware and instrumentation code.

Concerning runtime management of instrumentation overhead, the instrumentation layer used in the TAU CCA Tools approach can be disabled, but cannot be easily removed from call paths of instrumented components, like in the case of connector-based instrumentation. However, in the environment targeted by CCA this is not really an issue,

because the applications are used to perform (sometimes long) finite computations, but do not really provide services in a production environment typical for distributed applications.

Deployment integration

Even though the CCA platform does not define application deployment, the process of launching an application is in fact a part of deployment. The assembly of a CCA application is handled by a launcher program using the BuilderService port of a CCA runtime.

The proxy-based instrumentation technique used by TAU CCA Tools turns out to be somewhat unwieldy when considering the integration of the application instrumentation process with application deployment. Applying the technique to an application also requires the application architecture to be modified to accommodate the generated proxy component. The modification requires inserting additional components into the architecture, which is a more intrusive change than just renaming the components in case of wrapper-based instrumentation techniques. Consequently, the application instrumentation is not a transparent process, because it requires code changes in manually written launchers and launcher application assembly scripts. To our knowledge, the tools responsible for generating proxy components do not perform the required modifications to application architecture.

In contrast, the connector-based instrumentation technique employed in our approach is well integrated into the deployment process and provides a generic way for transparent instrumentation of component-based applications.

7.2.2 Other application-level approaches

Middleware-based and general application-level approaches are similar to component-based approaches, except they are targeted at "normal" applications, either local or distributed. Deployment of such applications is typically standardized at the level of application packaging and dependency resolution. Depending on the instrumentation technique used by a particular approach, an application must be instrumented either prior to packaging (and disseminated in instrumented form), or prior to execution after it has been installed onto a system.

In general, there is no benefit in comparing the deployment process of component based applications presented in this thesis with the deployment process of these applications, because they serve different purposes. While in case of component-based applications deployment serves both for installation and launch, in case of normal applications deployment primarily serves for installation of application binaries.

Wabash

Of the many features of Wabash [SMM00], the most relevant to this work is the support for performance measurement in application server objects [SMD00]. The Performance Instrumentation Module of the application instrumentation layer corresponds to the connector-based performance instrumentation and the measurement infrastructure proposed in this thesis.

The architecture of the measurement infrastructure employs similar concepts as the architecture of the Performance Instrumentation Module, but is more fine grained which makes it more flexible for use in different scenarios. The measurement infrastructure strictly separates between event sources, a subsystem for receiving event notifications and

taking snapshots of performance data, a subsystem for processing and storing event and performance data, and a subsystem for delivering the data outside the infrastructure.

The Performance Instrumentation Module is designed for a single type of performance events with semantics implicitly defined by the use of CORBA interceptors. In contrast, the concept of event sources employed in the design of the measurement infrastructure makes it independent of a particular instrumentation method. Dynamic registration of performance event sources along with predefined sources of simple performance events allows using the infrastructure both directly from application code and indirectly from performance instrumentation code.

Compared to connector-based instrumentation, the approach employed by Wabash is highly middleware specific, even though the concept of interceptors can be found in other kinds of middleware. With respect to managing performance measurement overhead, the instrumentation code in CORBA interceptors can be only made passive, because specifically in CORBA, the interceptors become a part of an ORB implementation and cannot be removed after an ORB has been initialized. Connectors, on the other hand, allow complete removal of instrumentation code from the call path of component interfaces for which no performance events should be emitted.

The design of the measurement infrastructure also allows spending less time in the application execution context, because event data processing can be performed independently of event processing. In case of Wabash, all event and performance data processing is performed inside the interceptor, with asynchronous processing supported only for storing event records to a log and providing aggregate statistical data to a zonal manager. In the measurement infrastructure, this would be the responsibility of the Data Delivery subsystem.

The event and data processing part of a Wabash interceptor is static with respect to the type of collected performance data. In contrast, the measurement infrastructure allows associating performance data with performance events independently of the event and event processing code. The association is defined by a user and can be created dynamically at runtime. In addition, the Performance Data subsystem of the infrastructure allows associating arbitrary performance data with a performance event.

StatWrapper

Even though due to lack of technical details concerning the access and collection of performance data the approach presented in [MC+99] is unsuitable for comparison with the work presented in this thesis, the analysis of the instrumentation techniques available in the context of CORBA-based application is precise and exhaustive and complements the interceptor-based approach employed by Wabash.

At that time, both approaches were fragile and specific to a particular CORBA implementation and were either underspecified (such as BOA), or not covered by the specification (TIE classes and interceptors). However, the concepts behind StatWraper and Wabash can be used even today, because the main concepts (object adaptor inheritance and request interceptors) are well defined by the specification and supported by most existing CORBA implementations.

TAU performance system

The TAU performance system [SM06, PRL07a] is probably the most relevant related work in the area of general application-level approaches, which can be also used with

component applications based on the CCA component system through its CCA Tools [MS+05, PRL05]. The TAU architecture has three layers covering the instrumentation, measurement, and analysis aspects of measurement-based performance analysis. Of those, only the first two layers are relevant to the work presented here, since the analysis of performance data is beyond the scope of this thesis.

Application instrumentation

This first layer of TAU architecture is responsible for performance instrumentation, which corresponds to the connector-based instrumentation presented in Chapter 5. The TAU instrumentation layer is generic in the respect that it can be used with applications in almost any form and includes both intrusive (source code modification, binary code patching) and non-intrusive (wrapper libraries, interpreter and virtual machine attachment) instrumentation techniques. The connector-based instrumentation proposed in this thesis is specifically targeted at component applications and aims to be generic with respect to different component systems.

In contrast to most instrumentation techniques supported by TAU, the connector-based instrumentation is both transparent and non-intrusive and does not depend on low-level properties of the execution platform, such as what processor type will be used to run the application code. Compared to proxy and wrapper based instrumentation techniques, there are no issues with integration into application architecture and life cycle management. The architecture-based connector model allows using enhanced component interfaces inside a connector, which may be used to convey additional information from the client to the server, such as global caller identification.

Performance data collection

The second layer of TAU architecture is responsible for tasks related to performance measurements, which corresponds to the measurement infrastructure presented in Chapter 4.

The performance events supported by both systems are similar, but TAU uses a dynamic event naming and lazy event registration (i.e. an event is registered with TAU after it first occurs), while the measurement infrastructure proposed in this thesis uses explicit event naming and event registration to provide strict separation between infrastructure configuration and infrastructure operation. This is to minimize the influence of the measurement infrastructure operations on the collected performance data. In addition, the measurement infrastructure supports the concept of event sources, which are entities providing multiple performance events. This concept is suitable for design-level instrumentation with an instrumentation entity corresponding e.g. to a single interface. Event source provide the measurement infrastructure with a control interface which allows the measurement infrastructure (and its users) to select which performance events should be generated by a particular event source.

Concerning performance data collection, TAU supports collecting performance data from different sources, such as processor hardware counters, selected system and process related counters, and even counters provided by application itself. However, the selection of performance data to be collected is made globally, and the sources of performance data (with the exception of application specific counters) are predefined. In contrast, the measurement infrastructure is designed to allow dynamic association of arbitrary performance data with individual performance events. The infrastructure uses a specialized subsystem to access performance data from various sources in a generic way.

The last aspect of measurement related tasks corresponds to storage and delivery of performance data. TAU supports two different measurement modes, one for storing event traces, and the other for storing aggregate performance metrics for one or more performance counters. With respect to data delivery, both measurement modes support exporting collected performance data in various data formats. The measurement infrastructure is designed to primarily store event traces along with associated performance data. Though not elaborated in detail in this thesis, the infrastructure is intended to support dynamic association of aggregate performance metrics with individual events. Delivery is also not elaborated in detail in the current infrastructure design, but the entities responsible for storing event traces and performance data in memory provide access to the store data, therefore a suitable data delivery method can be implemented on top of the infrastructure.

Chapter 8

Conclusion

8.1 Summary

The main motivation behind this work is a long-term goal to simplify application of measurement-based performance analysis techniques such as performance evaluation, performance monitoring, and regression benchmarking in the context of component-based applications so that ideally, they may be adopted as a standard development and engineering practice. We consider the main obstacle in adoption of such techniques to be a perceived high cost of obtaining and analyzing performance data contrasted with marginal short-term benefits.

In this work, we have proposed a generic approach to collection of performance data for heterogeneous component-based applications. When realized, the approach should lower both the perceived and the real costs of obtaining data for tracking and analyzing performance during development of component applications. In addition, easier access to performance data from real systems or their executable models should be also beneficial for model-based performance analysis methods, because it will simplify the experimental work necessary for construction of accurate application-specific performance models.

To better manage problem complexity, the main goal of the thesis has been split into three subgoals, each dealing with a particular aspect of the performance data collection process.

Goals and solutions

The first subgoal concerns the actual collection of various performance data related to application execution. The solution to the first subgoal has been described in Chapter 4 and is represented by a design of a generic measurement infrastructure which handles the tasks common to all performance measurement experiments, such as collecting and storing performance data related to application execution. Specifically though, the infrastructure provides generic access to different performance data sources and allows various performance data to be associated with generic performance events, which drive the performance data collection process.

The second subgoal concerns performance instrumentation of component-based applications, which is required to define and emit performance events related to application execution needed for operation of the measurement infrastructure. The solution to the second subgoal has been described in Chapter 5 and is represented by a design of connector-based application instrumentation technique which provides design-level performance events driving the collection of performance data within the measurement infrastructure. The instrumentation method utilizes an existing model of architecture-based connectors [Bur06] to enable non-intrusive, transparent, and generic performance instrumentation of component-based applications.

The third subgoal concerns transparent integration of the performance data collection process with the life cycle of heterogeneous component-based applications. The solution to the third subgoal has been described in Chapter 6 and is represented by a proposed integration of connectors and connector generation into deployment and configuration of distributed component-based applications defined by OMG [OMG06c]. The integration of connectors aims to enable using the deployment process even in context of heterogeneous component-based applications, with connectors serving to bridge the differences and incompatibilities among components from different component systems. Since the approach to deployment of heterogeneous applications is based on connectors, the connector-based performance instrumentation can be naturally integrated with the deployment process which makes it completely transparent to the developer or application deployer.

When combined, these solutions provide a conceptual and design foundation for the proposed generic approach to performance data collection suitable for application in the context of:

- heterogeneous applications, assembled from components based on different component systems,
- highly diverse environment, combining different hardware architectures, programming languages, operating systems, and middleware, and
- off-the-shelf software, distributed without application source code.

Each of the above solutions provides certain aspect of the proposed generic approach, which represents the most general case to which it can be applied. Depending on the realization of individual solutions, the approach can be used in more specific contexts without having to change the fundamental concepts. For example, if a deployment runtime does not support heterogeneous applications, the approach remains the same, except performance data collection is not supported for such applications.

Review of contributions

The main contribution of this work is a conceptual framework for a generic approach to obtaining design-level performance data from heterogeneous component-based applications. Mainly due to the amount of work needed for actual realization we do not provide an implementation of the presented approach. However, the designs presented in this thesis provide a foundation for incremental development of an infrastructure which will simplify application of measurement-based performance analysis techniques to component-based applications.

Secondary contributions of this work arise from the solutions to the individual subgoals of this thesis. Even though the above solutions have been designed for mutual interoperability in order to achieve the main goal of the thesis, the focus on generic design allows them to be used independently of each other to address a specific issue in a different context. With respect to the current status of the work, which will be discussed later, this also means that the realizations of the above solutions can be developed separately and still be individually useful.

Concerning performance data collection, the main contribution is the design of a generic measurement infrastructure and its performance data access layer. Different types of performance data are supported through a common abstraction represented by sensors. The sensors are provided by data source modules, which are responsible for providing the

sensors with names that allow them to be uniquely identified and associated with the respective data source module. Different data source modules provide sensors related to performance data available in different data sources.

Performance data represented by sensors are accessed through a measurement context, which captures a particular selection of sensors and decouples the operations needed to map sensors to the performance data they represent from the requests to obtain their values. This concept is essential for efficient access to performance data, because it allows data source modules responsible for the selected sensors to choose an optimal strategy for accessing the underlying data source prior to operations related to sampling and decoding sensor values.

The measurement infrastructure can use a measurement context to easily associate performance data with performance events without the need to evaluate the association at each occurrence of a performance event. The design of the remaining parts of the infrastructure captures the essential concepts needed to create a flexible measurement infrastructure that can be used either from performance instrumentation code or directly from applications. An important aspect captured in the infrastructure design is separation of event processing from storage of performance data, which allows storing the data asynchronously to application execution context.

Concerning application instrumentation, an important contribution is the concept of connector-based performance instrumentation, which provides a generic, high-level, compositional method for definition and generation of design-level performance events. This allows applying the performance data collection approach in a generic way to component models with support for architecture-based connectors.

Additional contribution is a connector reconfiguration mechanism which allows including or excluding a particular connector element from the call path of component business methods. This mechanism can be implemented within performance instrumentation elements, which may exclude themselves from execution when they are configured to provide no performance events. This in turn allows enabling and disabling the entire instrumentation layer at runtime, and letting the instrumented application run completely undisturbed by performance instrumentation. The mechanism is also generally applicable to other interception related functions that may be provided by connector elements.

Concerning heterogeneous deployment, an important contribution is the integration of connectors into the deployment process of component based applications defined by OMG [OMG06c]. Connectors mediate communication among components and can be used to bridge the differences between components from different component systems or components implemented in different programming languages. The integration of connectors into the deployment process thus enables deployment of heterogeneous component-based applications using a common, standard-based deployment process.

The ability to deploy heterogeneous applications in a unified way allows applying the proposed approach to performance data collection even to heterogeneous component-based applications, provided there is an implementation of the measurement infrastructure for a particular programming language as well as a connector generator able to produce performance instrumentation elements in that language.

8.2 Current Status

In this thesis, we have focused on providing a detailed description of the key concepts and abstractions necessary to realize the proposed approach, along with detailed design of the main parts of the solution. However, the scope of the work does not allow a single person to implement the entire infrastructure; therefore the implementation of the performance data collection framework for component applications is not yet available.

A proof-of-concept implementation of the performance data access layer has been created to aid in the design of the Performance Data subsystem of the measurement infrastructure. Similar proof-of-concept work has been done on the measurement infrastructure in context of the CoCoME project [BB+07], but due to the nature of the assignment, it was targeted for the Julia implementation of the Fractal component model, using Julia specific interception capabilities for performance instrumentation.

Even though these implementations are far from a prototype stage, they were instrumental in designing the architecture of the measurement infrastructure and its subsystems. They also provided insight on the true scope of the work and effort needed to create a fully functional implementation of the performance data collection framework. Since individual parts of the framework are individually useful, we have split the realization of the approach into several projects that are slowly materializing.

We have started with the deployment framework which represents an integration platform for the proposed approach. Deployment runtime adhering to the OMG Deployment and Configuration specification was designed and implemented by a graduate student as a master thesis project [Saf07]. The implementation closely follows the OMG specification, introducing only changes necessary to support deployment of heterogeneous applications. Different component systems are supported via plugins which provide the generic part of the runtime with unified interface for managing life cycle of components from a particular component system. So far, the system supports components in the Fractal component model. Support for the SOFA component model is pending the completion of the SOFA2 runtime.

Another master thesis project is currently underway and is concerned with the design and implementation of a deployment planner tool with support for integrating a connector generator into the planning stage of the deployment process. This includes assigning components to computational nodes in a target domain, as well as transforming a logical architecture of an application to its physical counterpart represented by a deployment plan that can be used to deploy an application into the above mentioned runtime.

No other projects are currently underway, but the scope of the remaining parts of the infrastructure is similar to that of the deployment part.

8.3 Future Work

The priority for future work is to bring the concepts presented in this work to life and evaluate the approach in practical applications. However, this requires implementing also the other parts of the framework. The measurement infrastructure needs to be split into two projects, one responsible for the Performance Data subsystem and the other for the rest of the infrastructure. Realization of the connector-based application instrumentation should fit into a single project.

The Performance Data subsystem has a low-level and high-level part. The low-level part provides access to data source interfaces that are not accessible from the implementation language of the high-level part. On top of these, data source modules for accessing performance data at least on the Linux, Windows, and Solaris operating systems will need to be implemented, including a special data source providing platform independent sensors. The high-level part, providing configuration and management functionality, will need to be implemented at least in Java, which is the native language of the SOFA and Fractal component systems which we primarily use in our research.

Additional research needs to be done to evaluate the usability of the performance data access layer, quantify the overhead it imposes on the system due to its generic design, and evaluate its scalability in multi-threaded environment, which is related to internal memory management and synchronization.

The other subsystems of the infrastructure, specifically the management, event processing, and data storage subsystems will have to provide at least the functionality described in the presented infrastructure design. This will allow using the infrastructure with the component systems preferred in our research. In future, the infrastructure will have to provide support for data aggregation to enable low-impact runtime monitoring of an application without the need to store data corresponding to individual event records.

The realization of connector-based performance instrumentation may appear simple, because the connector generator has been already implemented and enhanced to support more powerful code generation techniques based on the Stratego/XT program transformation language. Generating code of connector elements is therefore rather simple, even though there are issues to address such as providing client context to performance events generated by elements in server connector units. The complexity is hidden elsewhere. The connector-based instrumentation process is a bridging element between the measurement infrastructure and the infrastructure for deployment of heterogeneous component-based applications, both of which can be used separately. The realization of the connector-based instrumentation should therefore focus on smooth integration of the connector generation process (including instrumentation) into the deployment process, building on the work on the deployment planner currently underway.

Additional research in this area is related to generalization of the connector generation approach to generation of control parts of application components.

Finally, realization of the approach presented in this thesis will provide technical infrastructure for further research related to performance analysis based on experimental data from real systems, as well as research related to autonomic computing, such as monitoring and anomaly detection.

References

AA+06 Allan, B. A., Armstrong, R., Bernholdt, D. E., Bertrand, F., Chiu, K., Dahlgren, T. L., Damevski, K., Elwasif, W. R., Epperly, T. G. W., Govindaraju, M., Katz, D. S., Kohl, J. A., Krishnan, M., Kumfert, G., Larson, J. W., Lefantzi, S., Lewis, M. J., Malony, A. D., McInnes, L. C., Nieplocha, J., Norries, B., Parker, S. G., Ray, J., Shende, S., Windus, T. L., and Zhou, S. A Component Architecture for High-performance Scientific Computing. *Intl. Journal of High Performance Computing Applications*, 20(2):163-202, SAGE Publications, 2006.

AB83 Agrawal, S. C., Buzen, J. P. The Aggregate Server Method for Analyzing Serialization Delays in Computer Systems. *ACM Transactions on Computer Systems*, 1(2):116-143, ACM, May 1983.

AG+99 Armstrong, R., Gannon, D., Geist, A., Keahey, K., Kohn, S. R., McInnes, L., Parker, S. R., Smolinski, B. A. Toward a Common Component Architecture for High-Performance Scientific Computing. Proc. of the 8th IEEE Intl. Symposium on High-Performance Distributed Computing (HPDC 1999), Redondo Beach, CA, USA, IEEE CS Press, 1999.

ASF06 Apache Software Foundation. BCEL: Byte Code Engineering Library. *http://jakarta.apache.org/bcel*, June 2006.

ASF07 Apache Software Foundation. Apache Velocity Template Engine. *http://jakarta.apache.org/velocity*, 2007.

Bal02 Balek, D. Connectors in Software Architectures. Dissertation Thesis, Dept. of SW Engineering, Charles University, Prague, March 2002.

Bar05 Barroso, L. A. The Price of Performance. *Queue*, 3(7):48-53, ACM Press, September 2005.

BB+07 Bulej, L., Bures, T., Coupaye, T., Decky, M., Jezek, P., Parizek, P., Plasil, F., Poch, T., Rivierre, N., and Sery, O. CoCoME in Fractal. To appear as a chapter in Proceedings of the CoCoME project, LNCS, June 2007.

BB03 Bulej, L., and Bures, T. A Connector Model Suitable for Automatic Generation of Connectors. Tech. Report 2003/1, Dept. of SW Engineering, Charles University, Prague, 2003.

BB04 Bulej, L., and Bures, T. Addressing Heterogeneity in OMG D&C-based Deployment. Tech. Report 2004/7, Dept. of SW Engineering, Charles University, Prague, November 2004.

BB04 Bulej, L., and Bures, T. Addressing Heterogeneity in OMG D&C-based Deployment. Tech. Report 2004/7, Dept. of SW Engineering, Charles University, Prague, November 2004.

BB05 Bulej, L., and Bures, T. Using Connectors for Deployment of Heterogeneous Applications in the Context of OMG D&C Specification. Proc. of the 1st Intl. Conference on Interoperability of Enterprise Software and Applications

(INTEROP-ESA 2005), Geneva, Switzerland, Springer-Verlag, February 2005.

BB06a Bulej, L., and Bures, T. Addressing Static Execution Overhead in Connectors with Disabled Optional Features. Tech. Report 2006/6, Dept. of SW Engineering, Charles University, Prague, June 2006.

BB06b Bulej, L., and Bures, T. Eliminating Execution Overhead of Disabled Optional Features in Connectors. Proc. of the 3rd European Workshop on Software Architectures (EWSA 2006), Nantes, France, LNCS 4344, Springer-Verlag, September 2006.

BBT03 Buble, A., Bulej, L., and Tuma, P. CORBA Benchmarking: A Course With Hidden Obstacles. Proc. of the 2003 International Parallel & Distributed Processing Symposium (IPDPS 2003), Workshop on Performance Modeling, Evaluation and Optimization of Parallel and Distributed Systems (PMEOPDS 2003), Nice, France, IEEE CS, April 2003.

BC+06 Bruneton, E., Coupaye, T., Leclerq, M., Quéma, V., and Stefani, J.-B. The Fractal Component Model and Its Support in Java. *Software Practice and Experience*, special issue on Experiences with Auto-adaptive and Reconfigurable Systems. 36(11-12), 2006.

BCR94 Basili, V. R., Caldiera, G., and Rombach, H. D. Goal Question Metric Paradigm. *Encyclopedia of Software Engineering*, 1:528-532, John Wiley & Sons, 1994.

BD+00 Browne, S., Dongarra, J., Garner, N., Ho, G., and Mucci, P. A Portable Programming Interface for Performance Evaluation on Modern Processors. *Intl. Journal of High Performance Computing Applications*, 14(3):189-204, SAGE Publications, Fall 2000.

BDM02 Bernardi, S., Donatelli, S., and Merseguer, J. From UML Sequence Diagrams and Statecharts to Analysable Petri Net Models. Proc. of the 3rd Intl. Workshop on Software and Performance (WOSP 2002), Rome, Italy, ACM Press, July 2002.

BG98 Bernardo, M., and Gorrieri, R. A Tutorial on EMPA: A Theory of Concurrent Processes with Nondeterminism, Priorities, Probabilities and Time. *Theoretical Computer Science*, 202(1-2):1-54, Elsevier Science B.V., July 1998.

BH00 Buck, B., and Hollingsworth, J. K. An API for Runtime Code Patching. *Intl. Journal of High Performance Computing Applications*, 14(4):317-329, SAGE Publications, November 2000.

BHP06 Bures, T., Hnetynka, P., and Plasil, F. SOFA 2.0: Balancing Advanced Features in a Hierarchical Component Model. Proc. of the 4th ACIS Intl. Conf. on Software Engineering Research, Management and Applications (SERA 2006), Seattle, WA, USA, IEEE CS Press, August 2006.

BKT04 Bulej, L., Kalibera, T., and Tuma, P. Regression Benchmarking with Simple Middleware Benchmarks. Proc. of the 23rd International Performance Computing and Communications Conference (IPCCC 2004), International Workshop on Middleware Performance, Phoenix, AZ, USA, IEEE CS, April 2004.

BKT05 Bulej, L., Kalibera, T., and Tuma, P. Repeated Results Analysis for Middleware Regression Benchmarking. *Performance Evaluation*, 60(1-4):345-358, Elsevier B.V., May 2005.

BM03 Balsamo, S., Marzolla, M. Simulation Modeling of UML Software Architectures. Proc. of the 17th European Simulation Multiconference, Nottingham, UK, SCS

Europe, June 2003.

BP01 Balek, D., and Plasil, F. Software Connectors and Their Role in Component Deployment. Proc. of the 3rd IFIP Intl. Conference on New Developments in Distributed Applications and Interoperable Systems (DAIS 2001), Krakow, Poland, Kluwer, September 2001.

BP03 Bures, T., and Plasil, F. Composing Connectors of Elements. Tech. Report 2003/3, Dept. of SW Engineering, Charles University, Prague, May 2003.

BP04 Bures, T., and Plasil, F. Communication Style Driven Connector Configurations. Selected revised papers: Software Engineering Research and Applications (SERA 2004), LNCS 3026, Springer-Verlag, 2004.

Bur06 Bures, T. Generating Connectors for Homogeneous and Heterogeneous Deployment. Dissertation Thesis, Dept. of SW Engineering, Charles University, Prague, September 2006.

BZ98 Berrendorf, R., and Ziegler, H. PCL - The Performance Counter Library: A Common Interface to Access Hardware Performance Counters on Microprocessors. Tech. Report FZJ-ZAM-IB-9816, Central Institute for Applied Mathematics, Research Centre Jülich, Germany, October 1998.

BZM03 Berrendorf, R., Ziegler, H., and Mohr, B. PCL – The Performance Counter Library. *http://www.fz-juelich.de/zam/PCL*, January 2003.

CA+05 Chen, T., Ananiev, L., Tikhonov, A, Zhang, Y., and Shi, A. Linux Kernel Performance Project. *http://kernel-perf.sourceforge.net*, 2006.

Can06 Cantrill, B. Hidden in Plain Sight. *Queue*, 4(1):26-36, ACM Press, February 2006.

CCA03 CCA Forum. CCA specification. *http://www.cca-forum.org/specification*, 2003.

CCME07 Modelling Contest: Common Component Modelling Example (CoCoME). *http://agrausch.informatik.uni-kl.de/CoCoME*, 2007.

CG+04 Cavenet, C., Gilmore, S., Hillston, J., Kloul, L., and Stevens, P. Analyzing UML 2.0 Activity Diagrams in the Software Performance Engineering Process. Proc. of the 4th Intl. Workshop on Software and Performance (WOSP 2004), Redwood City, CA, USA, ACM Press, January 2004.

CGH04 Cai, Y., Grundy, J., Hosking, J. Experiences Integrating and Scaling a Performance Test Bed Generator with an Open Source CASE Tool. Proc. of the IEEE Intl. Conf. on Automated Software Engineering (ASE 2004), September 2004

CM00 Cortelessa, V., and Mirandola, R. Deriving a Queuing Network based Performance Model from UML Diagrams. Proc. of the 2nd Intl. Workshop on Software and Performance (WOSP 2000), Ottawa, Canada, ACM Press, September 2000.

CSL04 Cantrill, B. W., Shapiro, M. W., and Leventhal, A. H. Dynamic Instrumentation of Production Systems. Proc. of the USENIX Annual Technical Conference (USENIX 2004), Boston, MA, USA, USENIX, July 2004.

DCU07 Distributed Systems Research Group, Charles University. Middleware Benchmarking Project. *http://dsrg.mff.cuni.cz/benchmarking*, June 2007.

DE+05 Dahlgren, T. L., Epperly, T. G. W., Kumfert, G., and Leek, J. Babel Users' Guide. Version 0.10.4, LLNL, Livermore, CA, 2005.

DHH01 DeRose, L., Hoover, T., Hollingsworth, J. K. The Dynamic Probe Class Library – An Infrastructure for Developing Instrumentation for Performance Tools. Proc. of the 15th International Parallel and Distributed Processing Symposium (IPDPS 2001), San Francisco, CA, USA, IEEE CS Press, April 2001.

Dmi03 Dmitriev, M. Design of JFluid: Profiling Technology and Tool Based on Dynamic Byte-code Instrumentation. Tech. Report 2003-0820, Sun Microsystems, 2003.

Dmi04 Dmitriev, M. Profiling Java Applications Using Code Hot-swapping and Dynamic Call Graph Revelation. Proc. of the 4th Intl. Workshop on Software and Performance (WOSP 2004), January 2004.

DOC07 Distributed Object Computing Group. Continuous Metrics for ACE+TAO+CIAO. *http://www.dre.vanderbilt.edu/Stats*, 2007.

DPE04 Denaro, G., Polin, A., Emmerich, W. Early Performance Testing of Distributed Software Applications. Proc. of the 4th Intl. Workshop on Software and Performance (WOSP 2004), Redwood City, CA, USA, ACM Press, January 2004.

DVV01 De Jonge, M., Visser, E., and Visser, J. XT: A Bundle of Program Transformation Tools. Proc. of Workshop on Language Descriptions, Tools and Applications (LDTA 2001), Genova, Italy, April 2001 .

EF06 Eclipse Foundation. AspectJ project. *http://www.eclipse.org/aspectj*, November 2006.

EF07a Eclipse Foundation. Eclipse – An Open Development Platform. *http://www.eclipse.org*, 2007.

EF07b Eclipse Foundation. Eclipse Test and Performance Tools Project. *http://www.eclipse.org/tptp*, 2007.

FJ+05 Fahringer, T., Jugravu, A., Pllana, S., Prodan, R., Seragiotto, C., and Truong, H.-L. ASKALON: A Tool Set for Cluster and Grid Computing. *Concurrency and Computation: Practice and Experience*, 17(2-4):143-169, John Wiley & Sons, February 2005.

GB05a Galik, O., and Bures, T. Handling Heterogeneity in Connector Generation. Technical Report 2005/2, Dept. of SW Engineering, Charles University, Prague, June 2005.

GB05b Galik, O., and Bures, T. Generating Connectors for Heterogeneous Deployment. Proc. of the 5th Intl. Workshop on Software Engineering and Middleware (SEM 2005), Lisbon, Portugal, ACM Press, September 2005.

GBC06 Glassbox Corp. Glassbox: Troubleshooting Agent for Java Applications. *http://www.glassbox.com/glassbox*, 2006.

GH+04 Gilmore, S., Hillston, J., Kloul, L., and Ribaudo, M. Software Performance Modeling Using PEPA Nets. Proc. of the 4th Intl. Workshop on Software and Performance (WOSP 2004), Redwood City, CA, USA, ACM Press, January 2004.

GH+95 Götz, N., Hermanns, H., Herzog, U., Mertsiotakis, V., and Rettelbach, M. Stochastic Process Algebras. Quantitative Methods in Parallel Systems, Esprit Basic Research Series 18, Springer-Verlag, 1995.

GK05 Gilmore, S., Kloul, L. A Unified Tool for Performance Modeling and Prediction. *Reliability Engineering and System Safety*, 89(1), Elsevier B.V., July 2005.

GP02 Gu, G., and Petriu, D. C. XSLT Transformation from UML Models to LQN

Performance Models. Proc. of the 3rd Intl. Workshop on Software and Performance (WOSP 2002), Rome, Italy, ACM Press, July 2002.

GP05 Gu, G., and Petriu, D. C. From UML to LQN by XML Algebra-based Model Transformations. Proc. of the 5th Intl. Workshop on Software and Performance (WOSP 2005), Palma de Mallorca, Spain, ACM Press, July 2005.

GW+04 Gerndt, M., Wismüller, R., Balaton, Z., Gombás, G., Kacsuk, P., Németh, Z., Podhorszki, N., Truong, H-L., Fahringer, T., Bubak, M., Laure, E., and Margalef, T. Performance Tools for the Grid: State of the Art and Future. White paper of the working group on Automatic Performance Analysis: Real Tools (APART), version 1.0, January 2004.

Hil96 Hillston, J. A Compositional Approach to Performance Modeling. Cambridge University Press, 1996.

HM+05 Huck, K., Malony, A. D., Bell, R., and Morris, A. Design and Implementation of a Parallel Performance Data Management Framework. Proc. of the International Conference on Parallel Processing (ICPP 2005), Oslo, Norway, IEEE CS Press, June 2005.

Hof05 Hoffman, B. Monitoring, at Your Service. *Queue*, 3(10):34-43, ACM Press, December 2005.

HWR99 Hrischuk, C. E., Woodside, C. M., and Rolia, J. A. Trace Based Load Characterization for Generating Software Performance Models. *IEEE Transactions on Software Engineering*, 25(1), IEEE CS, January 1999.

Jae05 Jaeger, A. GCC Performance Tracking. *http://www.suse.de/~aj/SPEC*, 2005.

KBT04 Kalibera, T., Bulej, L., and Tuma, P. Generic Environment for Full Automation of Benchmarking. Proc. of Net.ObjectDays 2004, 1st International Workshop on Software Quality (SOQUA 2004), Erfurt, Germany, tranSIT GmbH; also in Testing of Component-Based Systems and Software Quality, LNI 58, GI E.V., September 2004.

KBT05a Kalibera, T., Bulej, L., and Tuma, P. Benchmark Precision and Random Initial State. Proc. of the 2005 International Symposium on Performance Evaluation of Computer and Telecommunication Systems (SPECTS 2005), Cherry Hill, NJ, USA, SCS, July 2005.

KBT05b Kalibera, T., Bulej, L., and Tuma, P. Quality Assurance in Performance: Evaluating Mono Benchmark Results. Proc. of the 2nd Intl. Workshop on Software Quality (SOQUA 2005), Erfurt, Germany, LNCS 3712, Springer-Verlag, September 2005.

KBT05c Kalibera, T., Bulej, L., and Tuma, P. Automated Detection of Performance Regressions: The Mono Experience. Proc. of the 13th IEEE Intl. Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS 2005), Atlanta, GA, USA, IEEE CS, September 2005.

KC03 Kephart, J. O., and Chess, D. M. The Vision of Autonomic Computing. *IEEE Computer*, January 2003 .

KH+01 Kiczales, G., Hilsdale, E., Hugunin, J., Kersten, M., Palm, J., and Griswold, W. Getting Started with AspectJ. *Communications of ACM*, 44(10):59-65, ACM Press, October 2001.

KL+05 Kalibera, T., Lehotsky, J., Majda, D., Repcek, B., Tomcanyi, M., Tomecek, A., Tuma,

P., Urban, J. Automated Benchmarking and Analysis Tools. Proc. of the 1st Intl. Conference on Performance Evaluation Methodologies and Tools (VALUETOOLS 2006), Pisa, Italy, ACM Press, October 2006.

KP00 King, P., and Pooley, R. Derivation of Petri Net Performance Models from UML Specification of Communications Software. Proc. of the 11th Intl. Conference on Tools and Techniques for Computer Performance Evaluation (TOOLS 2000), Schaumburg, IL, USA, 2000.

Lau05 Laudon, J. Performance/Watt: The New Server Focus. *Computer Architecture News*, 33(4):5-13, Special Issue: dasCMP'05, ACM Press, November 2005.

LBS05 Leicher, A., Busse, S., and Süß, J. G. Analysis of Compositional Conflicts in Component-Based Systems. Proc. of the 4th Intl. Workshop on Software Composition (SC 2005), Edinburgh, Scotland, LNCS 3628, Springer-Verlag, April 2005 .

LC+00 Lindlan, K., Cuny, J., Malony, A. D., Shende, S., Mohr, B., Rivenburgh, R., and Rasmussen, C. A Tool Framework for Static and Dynamic Analysis of Object-Oriented Software with Templates. Proc. of the 2000 ACM/IEEE Conference on Supercomputing (SC 2000), Dallas, TX, USA, IEEE CS, November 2000.

LFG04 Liu, Y., Fekete, A., and Gorton, I. Predicting the Performance of Middleware-based Applications at the Design Level. Proc. of the 4th Intl. Workshop on Software and Performance (WOSP 2004), Redwood City, CA, USA, ACM Press, January 2004.

LG05 Liu, Y., and Gorton, I. Performance Prediction of J2EE Applications Using Messaging Protocols. Proc. of Intl. Conference on Component Based Software Engineering (CBSE 2005), 2005.

LMC02 Lopez-Grao, J. P., Merseguer, J., and Campos, J. From UML Activity Diagrams to Stochastic Petri Nets: Application to Software Performance Analysis. Proc. of the 17th Intl. Symposium on Computer and Information Sciences, Orlando, FL, USA, CRC Press, October 2002.

LNL04 Lawrence Livermore National Laboratory. Babel homepage. *http://www.llnl.gov/CASC/components/babel.html*, January 2004.

MC+95 Miller, B. P., Callaghan, M. D., Cargille, J. M., Hollingsworth, J. K., Irvin, R. B., Karavanic, K. L., Kunchithapadam, K., and Newhall, T. The Paradyn Parallel Performance Measurement Tool. *IEEE Computer*, 28(11):37-46, IEEE CS Press, November 1995.

MC+99 McGregor, J. D., Cho, I., Malloy, B. A., Curry, E. L., Hobatr, C. Collecting Metrics for CORBA-Based Distributed Systems. *Empirical Software Engineering*, 4(3):217-240, Springer-Verlag, September 1999.

MF03 Mohr, B., and Wolf, F. KOJAK - A Tool Set for Automatic Performance Analysis of Parallel Applications . Proc. of the 9th Intl. Conference on Parallel and Distributed Computing (EURO-PAR 2003), Klagenfurt, Austria, September 2003.

MM02 Mos, A., and Murphy, J. Performance Management in Component-Oriented Systems Using a Model Driven Architecture Approach. Proc. of the 6th IEEE Intl. Enterprise Distributed Object Computing Conference (EDOC 2002), Lausanne, Switzerland, September 2002.

Mos04 Mos, A. A Framework for Adaptive Monitoring and Performance Management of Component-Based Enterprise Applications. Dissertation Thesis, School of Electronic Engineering, Dublin City University, August 2004.

MP06 Mozilla Project. Quality Assurance: Performance Testing. *http://wiki.mozilla.org/MozillaQualityAssurance:Performance_Testing*, November 2006.

MPI94 MPI Forum. MPI: A Message Passing Interface Standard. *Intl. Journal of Supercomputer Applications*, 8(3/4):159-416, Special Issue on MPI, 1994.

MS+05 Malony, A. D., Shende, S., Trebon, N., Ray, J., Armstrong, R., Rasmussen, C., and Sottile, M. Performance Technology for Parallel and Distributed Component Software. *Concurrency and Computation: Practice and Experience*, 17(2-4):117-141, John Wiley & Sons, February 2005.

OMG01 Object Management Group. Model Driven Architecture. *http://www.omg.org/cgi-bin/doc?ormsc/01-07-01*, July 2001.

OMG04 Object Management Group. Common Object Request Broker Architecture: Core Specification. Version 3.0.3, *http://www.omg.org/cgi-bin/doc?formal/04-03-12*, March 2004.

OMG06a Object Management Group. Meta Object Facility Core. Version 2.0, *http://www.omg.org/cgi-bin/doc?formal/2006-01-01*, January 2006.

OMG06b Object Management Group. CORBA Component Model. Version 4.0, *http://www.omg.org/cgi-bin/doc?formal/06-04-01*, April 2006.

OMG06c Object Management Group. Deployment and Configuration of Component-based Distributed Applications Specification. Version 4.0, *http://www.omg.org/cgi-bin/doc?formal/06-04-02*, April 2006.

OMG99 Object Management Group. White Paper on Benchmarking. Version 1.0, *http://www.omg.org/cgi-bin/doc?bench/99-12-01*, December 1999.

OWC04 ObjectWeb Consortium. LeWYS is Watching Your System. *http://lewys.objectweb.org*, April 2004.

OWC06 ObjectWeb Consortium. ASM: Java Bytecode Manipulation Framework. *http://asm.objectweb.org*, November 2006.

OWC07a ObjectWeb Consortium. Fractal Component Model. *http://fractal.objectweb.org*, 2007.

OWC07b ObjectWeb Consortium. SOFA Component Model. *http://sofa.objectweb.org*, 2007.

OWC07c ObjectWeb Consortium. Dream: A Component-Based Communication Framework. *http://dream.objectweb.org*, June 2007.

OWC07d ObjectWeb Consortion. Julia: Reference Implementation of the Fractal Component Model. *http://fractal.objectweb.org/julia*, May 2007.

PAPI Performance Application Programming Interface. *http://icl.cs.utk.edu/papi*, 2007.

PBJ98 Plasil, F., Balek, D., and Janecek, R. SOFA/DCUP: Architecture for Component Trading and Dynamic Updating. Proc. of 4th International Conference on Configurable Distributed Systems, Annapolis, MA, USA, IEEE CS Press,

May 1998.

PM+05 Prochazka, M., Madan, A., Vitek, J., and Liu, W. RTJBench: A Real-Time Java Benchmarking Framework. *Studia Informatica Universalis Journal*, 4(1), March 2005.

PRL05 Performance Research Lab, University of Oregon. TAU's CCA Tools. *http://www.cs.uoregon.edu/research/tau/cca*, June 2005.

PRL07a Performance Research Lab, University of Oregon. TAU – Tuning and Analysis Utitlies. *http://www.cs.uoregon.edu/research/tau*.

PS02 Petriu, D. C., and Shen, H. Applying the UML Performance Profile: Graph Grammar-based Derivation of LQN Models from UML Specifications. Proc. of the 12th Intl. Conference on Modeling Tools and Techniques for Computer and Communication System Performance Evaluation, London, UK, LNCS 2324, Springer-Verlag, April 2002.

PTP00 Prochazka, M., Tuma, P., and Pospisil, R. Enterprise JavaBeans Benchmarking. Tech. Report 2000/4, Dept. of SW Engineering, Charles University, Prague, 2000.

RCJ00 Research Centre Jülich. Report on State of the Art in Automatic Performance Analysis. Deliverable of working group on Automatic Performance Analysis: Real Tools (APART), February 2000.

RFC1157 IETF Network Working Group. Simple Network Management Protocol (SNMP), RFC 1157, *http://www.faqs.org/rfcs/rfc1157.html*, May 1990.

Rog05 Rogers, D. Lessons from the Floor. *Queue*, 3(10):26-32, ACM Press, December 2005.

RS95 Rolia, J. A., and Sevcik, K. C. The Method of Layers. *IEEE Transactions on Software Engineering*, 21(8):689-700, IEEE CS, August 1995.

RT+04 Ray, J., Trebon, N., Shende, S., Armstrong, R., and Malony, A. D. Performance Measurement and Modeling of Component Applications in a High Performance Computing Environment: A Case Study. Proc. of the International Parallel and Distributed Processing Symposium (IPDPS 2004), 2004.

Saf07 Safrata, P. Infrastructure for Deployment of Heterogeneous Component-based Applications. Master Thesis, Dept. of Software Engineering, Charles University, Prague, 2007.

SG98 Spitznagel, B., and Garlan, D. Architecture-based Performance Analysis. Proc. of the 10th International Conference on Software Engineering and Knowledge Engineering, Knowledge Systems Institute, 1998.

SI+04 Simeoni, M., Inverardi, P., Di Marco, A., and Balsamo, S. Model-Based Performance Prediction in Software Development: A Survey. *IEEE Transactions on Software Engineering*, 30(5):295-310, IEEE CS, 2004.

Sim07 Simakov, P. Linguine Watch: Library for real-time monitoring of Java software applications. *http://www.softwaresecretweapons.com/jspwiki/linguinewatch*, January 2007.

SM+03 Shende, S., Malony, A. D., Rasmussen, C., and Sottile, M. A Performance Interface for Component-based Applications. Proc. of the International Parallel and Distributed Processing Symposium (IPDPS 2003), Workshop on Performance

Modeling, Evaluation and Optimization of Parallel and Distributed Systems (PMEO-PDS 2003), Nice, France, IEEE CS Press, April 2003.

SM06 Shende, S., and Malony, A. D. The TAU Parallel Performance System. *Intl. Journal of High Performance Computing Applications*, 20(2):287-331, SAGE Publications, 2006.

SMA01 Shende, S., Malony, A. D., and Ansell-Bell, R. Instrumentation and Measurement Strategies for Flexible and Portable Empirical Performance Evaluation. Proc. of Intl. Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA 2001), CSREA, June 2001.

SMD00 Sridharan, B., Mathur, A. P., Dasarathy, B. On Building Non-Intrusive Performance Instrumentation Blocks for CORBA-Based Distributed Systems. Proc. of the 4th Intl. Computer Performance and Dependability Symposium (CPDS 2000), Schaumburg, IL, USA, IEEE CS, March 2000.

SMM00 Sridharan, B., Mundkur, S., Mathury, A. P. Non-Intrusive Testing, Monitoring and Control of Distributed CORBA Objects . Proc. of the 33rd Intl. Conference on Technology of Object-Oriented Languages and Systems (TOOLS 2000), France, IEEE CS, June 2000.

Sou07 Souza, S. JAMon: A Monitoring API. *http://jamonapi.sourceforge.net*, April 2007.

STRXT Stratego/XT, *http://www.stratego-language.org*, 2007.

SUN01 Sun Microsystems, Inc. Java Remote Method Invocation Specification. 2001.

SUN03 Sun Microsystems, Inc. Enterprise Java Beans. Version 2.1, November 2003.

SUN06a Sun Microsystems, Inc. J2EE Management Specification. Version 1.1, June 2006.

SUN06b Sun Microsystems, Inc. Java Management Extensions Specification. Version 1.4, November 2006.

SW00 Smith, C. U., and Williams, L. G. Performance and Scalability of Distributed Software Architectures: An SPE Approach. *Journal of Parallel and Distributed Computing Practices*, 3(4), SWPS, December 2000.

SW02 Smith, C. U., and Williams, L. G. Performance Solutions. Addison-Wesley. 2002.

TB01 Tuma, P., and Buble, A. Open CORBA Benchmarking. Proc. of the 2001 Intl. Symposium on Performance Evaluation of Computer and Telecommunication Systems (SPECTS 2001), Orlando, FL, USA, SCS, July 2001.

TB06 Tuma, P., and Bulej, L. Xampler: Application for Detailed Benchmarking of Middleware. *http://dsrg.mff.cuni.cz/ccpsuite*, Distributed Systems Research Group, Charles University, December 2006.

TF03 Truong, H.-L., and Fahringer, T. SCALEA: A Performance Analysis Tool for Parallel Programs. *Concurrency & Computation: Practice & Experience*, 15(11-12):1001-1025, John Wiley & Sons, August 2003.

TM+06 Trebon, N., Morris, A., Ray, J., Shende, S., and Malony, A. D. Performance Modeling of Component Assemblies. *Concurrency and Computation: Practice and Experience*, 19(5):685-696, John Wiley & Sons, October 2006.

WK98 Weinreich, R., and Kurschl, W. Dynamic Analysis of Distributed Object-Oriented Applications. Proc. of the 31st Annual Hawaii International Conference on System Sciences (HICSS 1998), Kohala Coast, HI, USA, IEEE CS Press,

January 1998.

Woo89 Woodside, C. M. Throughput Calculation for Basic Stochastic Rendezvous Networks. *Performance Evaluation*, 9(2):143-160, Elsevier B.V., April 1989.

WP+05 Woodside, C. M., Petriu, D. C., Petriu, D. B., Shen, H., Israr, T., and Merseguer, J. Performance by Unified Model Analysis (PUMA). Proc. of the 5th Intl. Workshop on Software and Performance (WOSP 2005), Palma de Mallorca, Spain, ACM Press, July 2005.

WS02 Williams, L. G., and Smith, C. U. PASA: A Method for the Performance Assessment of Software Architecture. Proc. of the 3rd Intl. Workshop on Software and Performance (WOSP 2002), Rome, Italy, ACM Press, July 2002.

WW04 Wu, X., and Woodside, C. M. Performance Modeling from Software Components. Proc. of the 4th Intl. Workshop on Software and Performance (WOSP 2004), Redwood City, CA, USA, ACM Press, January 2004.

ZL+95 Zielinski, K., Laurentowski, A., Szymaszek, J., and Uszok, A. A Tool for Monitoring Software-Heterogeneous Distributed Object Applications. Proc. of the 15th International Conference on Distributed Computing Systems (ICDCS 1995), Vancouver, BC, Canada, IEEE CS Press, June 1995.