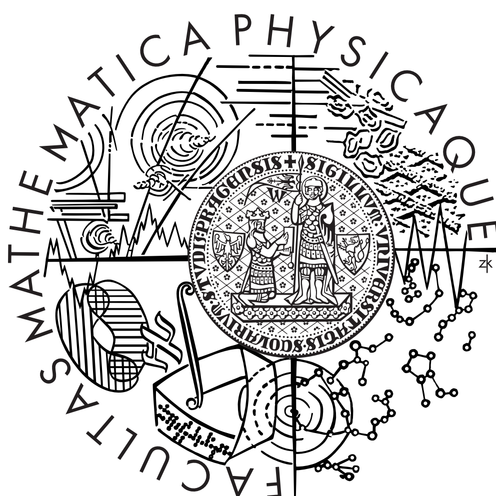
Charles University, Prague, Czech Republic
Faculty of Mathematics and Physics

MASTER THESIS



Pavel Ondroušek

Network Repository for Performance Evaluation Results

Department of Software Engineering

Supervisor: Doc. Ing. Petr Tůma, Dr.
Computer Science Program, Software Systems

2006

I would like to thank my supervisor, Petr Tůma, for his expert advice, interesting observations, brilliant ideas and great patience, which helped me to finish the thesis.

Many thanks also to my sister for proofreading the thesis.

I declare that I have written this master thesis myself, using only the referenced sources. I give my consent with lending the thesis.

Prague, December 15, 2006 Pavel Ondroušek

Contents

1 Introduction **4**
1.1 Goals 4
1.2 Definitions 4
1.3 Overview of thesis structure 5

2 Requirements analysis **6**
2.1 Storage and Plotting 6
2.2 Original form 8
2.3 Parsing 9
2.4 Configuration description 11
2.5 Evaluation 12
2.6 Plotting 14
2.7 Security 16
2.8 Summary 16

3 Repository design **17**
3.1 Storage system 17
3.2 Evaluation 20
3.3 Plotting 22

4 Benchmark model **24**
4.1 Content elements 24
4.2 Description elements 28
4.3 Evaluation elements 29

5 Implementation details **31**
5.1 Repository architecture 31
5.1.1 Data storage tier 31
5.1.2 Application tier 32
5.1.3 Client tier 32
5.2 Technology overview 32
5.3 Object model 32
5.3.1 Repository Server 33
5.4 Extending repository 35

6 Related works **38**

7 Conclusion **40**

Appendices **42**

Abstract

Název práce: *Network Repository for Performance Evaluation Results*
Autor: *Pavel Ondroušek*
Katedra: *Katedra softwarového inženrýství KSI*
Vedoucí diplomové práce: *Doc. Ing. Petr Tůma, Dr.*
Email vedoucího: *petr.tuma@mff.cuni.cz*

Abstrakt: *Pro měření výkonu softwarových systemů se používá benchmarkovaní. Benchmarkovaní vytváří velké množství dat, které je třeba ukládat, zpracovávat a vyhodnocovat. Network Repository slouží jako úložiště výsledků vznikajících během benchmarkovani. Cílem diplomové práce je navrhnout a naimplementovat datové úložiště podporující různé druhy výsledků s možností konfigurace jejich formátu. Uložené výsledky je možné dále zpracovávat, vyhodnocovat a používat jako zdroj pro generovaní grafických výstupů.*

Součástí diplomové práce je návrh a implementace obecného frameworku pro ukládání a vyhodnocovaní dat benchmarků a vytvoření vzorové konfigurace pro vybrané formáty dat výsledků benchmarků.

Klíčová slova: *benchmark, úložiště, grafy*

Abstract

Title: *Network Repository for Performance Evaluation Results*
Author: *Pavel Ondroušek*
Department: *Department of Software Engineering, KSI*
Supervisor: *Doc. Ing. Petr Tůma, Dr.*
Supervisor's email address: *petr.tuma@mff.cuni.cz*

Abstract: *For performance evaluation of software systems, benchmarking is used. Benchmarking generates a large amount of output data, which is necessary to store, process and evaluate it. Network Repository serves as a repository of benchmark results. The goal of the master thesis is to design and implement the data storage with a support of various result types with the possibility of the format configuration. Stored results can be processed, evaluated, or used as a source for plotting.*

As part of the master thesis, the design and implementation of a general framework for benchmark storage and evaluation is developed and a sample configuration for the selected benchmark result data format is established.

Keywords: *benchmark, repository, plotting*

1 Introduction

Software development is closely connected with software testing and performance evaluation. One of the ways to evaluate software performance is software benchmarking. Software benchmarking is a recurrent process producing a large amount of output data. Along with the growing complexity of the developed software, the amount and complexity of benchmark outputs is increasing, which implies the growing requirements for benchmark output storage. To support a variety of result formats, a configurable repository should be created. The repository should also provide basic evaluation and visualization facilities.

1.1 Goals

The goal of this thesis is to analyze various existing result formats [5-10] and, based on the analysis, to design and implement a network repository for performance evaluation results which will be accessible from various platforms. The repository must be able to support result data in many ad hoc formats, to preserve the original form of results with a detailed configuration description and information about the author of a result. The variability of the result formats makes it not convenient to use the existing applications, such as relational or object databases.

The repository must be able to provide a configurable persistent storage, which allows the user to store various result types. The stored results must be easily accessible for the user. The repository should provide the basic visualization facilities; for each supported result type, a predefined set of basic plot types is supported.

The repository interface must be general enough to be accessible from various benchmarking applications and frameworks, independently from the used programming language or the result data format. Both on-line storage by submitting data directly from a running application and offline storage by submitting the result files should be also supported.

1.2 Definitions

The output from performance evaluation software is called a *result*. The benchmark results contain various definitions of timestamps and the associated data. To be similar to the common statistical terminology a timestamp or another measured value is called an *observation*. A set of observations is called a *sample*. The main features of benchmarks are repetitiveness and associations. Each benchmark result can contain multiple samples with different *benchmark setup*, which contains some benchmark-specific features, such as the count of executed threads, the count of object instances, the data type, the data size, and the repetition count. There is a need to describe the relation between the sample and the benchmark setup. The relation is called an *association*. Each stored result is also supplied with a *configuration description*. The configuration description includes the hardware and software setup, the running processes, the processor usage, and the memory consumption. These settings are typical parameters for searching. A configuration description entry is called a *property*. During an experiment, the benchmark setup differed for each measured sample, while the configuration description remained the same.

1.3 Overview of thesis structure

The thesis consist of a text-based work and a repository implementation. The text work consists of five sections - requirement analysis, repository design, benchmark model, repository architecture and related work. The benchmark model section can also serve as a documentation for the benchmark model description creation.

The XML Schema definition of repository outputs is contained in Appendix A and Appendix B. In Appendix C, the sample benchmark model description is established. The CD-ROM detailed content is specified in Appendix D.

The implementation is contained on a CD-ROM. The CD-ROM content consists of the Network Repository source code, the programming documentation in the Javadoc format, the binary distribution of Network Repository, the administration guide and sample data. The administration guide contains the installation manual and the basic configuration settings.

2 Requirements analysis

The repository design and implementation are influenced by many requirements that originate from the benchmark usage and the results structure. These requirements are partially contained in the assignment and they have also partially emerged during the analysis of benchmark processes and benchmark results. In this chapter, the main problems of implementation are analyzed. For each problem, a detailed description, possible choices and a selected solution are created. In the problem description, the quotation from the assignment are highlighted.

Assignment

Design and implement a network repository for performance evaluation results. The repository should meet the following criteria:

- Customizable support for multiple result formats including plain text and XML formats. As a proof of concept, support for at least the Sampler, Xampler, RUBiS, ECperf, TAO, OVM benchmarks.
- Customizable support for multiple display formats including box and whisker plots, density plots, history plots. As a proof of concept, support for at least the Sampler, Xampler, TAO, OVM display formats.
- Browsable via web and accessible via web services interface.
- Ability to store configuration description alongside results.
- Ability to export results in original result formats.
- Support for various levels of result trust.
- Implementation environment Linux with PHP and Java, no databases.

Benchmarking

The benchmarking process is usually very sensitive to any operation overhead, such as a result evaluation or a complex calculation. The benchmark authors, therefore, require that the benchmark generates the simplest possible output, because the overhead of a time-consuming operation can misrepresent the obtained results. The generated data have a fixed structure depending on the type of the measured feature. A human readable summary is also often present, as the result data are very large for a quick overview. Thus the benchmark variability leads to the existence of many ad hoc result formats.

2.1 Storage and Plotting

The repository should provide a *customizable support for multiple result formats, including plain text, binary and XML formats*. The repository should provide a *customizable support for multiple display formats, including box and whisker plots, density plots, and history plots*.

Details

The aim of the thesis is to create a repository that will support a large amount of result formats and generate the possible plots for all of them. The benchmark results are usually large and complex, and there is a need of preprocessing the data before generating a plot. The implementation should provide the user with the facility to create a specified plot for each stored result or a set of results.

Choices

The simplest choice is to create a plotting component for each supported benchmark type. Considering the count of supported benchmark types M and the count of supported plots N the count of created components is equal to $M.N$. This solution would lead to the creation of M new components with adding a new plot or to the creation of N new components with adding a new benchmark type. So this solution is unacceptable.

The problem can be reduced by defining a *common data format*. The common data format is an internal repository data structure, which is general enough to cover the possible result types. While processing, all supported benchmark results are converted to this data format without any information loss. The problem of a large amount of plotting components is solved using the common data format as a resource for the plotting process.

The common data format should reflect the structure of benchmark results. First, the matrix format will be considered. In the matrix format, the observation are stored in arrays. A sample consists of an array of observation of arrays. A benchmark result is represented with an array of sample arrays. The advantage of the matrix format is an access to the memory and easy lookup of a concrete value. A problem occurs when more samples with different observation and sample counts are processed together. There is also no effective way to associate the corresponding benchmark setup with the processed observations. The fragmentation of various benchmark types enforces using a data structure more general than a matrix. After the analysis of example result structures, a tree representation of data is chosen. The tree structure is easy to represent in memory, moreover, there is a lot of ways to split and merge two or more trees. The disadvantage of the tree representation is slower lookup of a concrete value.

Decision

In order to minimize the number of the created components, a common data format is established. The structure of the common data format is based on a tree representation. The tree is able to preserve the original structure of samples together with an associated benchmark setup or configuration description. There are two types of nodes, decision nodes and leaves. The result observations are stored in the leaves as a collection of values. A concrete value is identified by an ordinal number. Decision nodes represent the associations between the benchmark setup and the result data. An association is stored as a string value and corresponds to the node name. The parsing process objective is the correct coverage of the associations.

2.2 Original form

The repository should provide *the ability to export results in the original result format*. During the realization, *no database* should be used.

Details

Typically, many tools and utilities working with the original result format exist, therefore there is a need of the facility to return the results without any change. Keeping the stored results in the original format is also important because of the credibility of the repository. The users often think there is an information loss if the returned data are in a different format.

At the same time there is a need to design a storage mechanism independent of any existing database. In connection with the variability of results and the requirement to preserve the original format, the usage of a database is not as effective as in case of storing in an internal data format with a fixed structure.

Choice

The storage realization is based on the requirements that the benchmark results are stored in the original format, the directory structure is also preserved. The problem of the required storage mechanism is low performance. There is a need to design basic database facilities on top of the repository storage structure.

The repository storage structure should be well arranged, so it is necessary to design a directory structure that would help with the orientation in the repository. The first considered characteristics is the repetitiveness of the structure for the stored results of the same benchmark type. For each benchmark type, the requirements for searching are the same; it is useful to group benchmarks of the same type together. Thus, there is a common parent directory for each supported benchmark type. What remains as a problem is sorting the stored results with the same type. There is a lot of possibilities of creating a useful directory structure. The results can be differentiated by their content. For every association, a corresponding directory can be created, and the result can be stored in it. This solution is convenient especially for the repository administrators if there is a need of manual searching for a particular benchmark and the required parameters are known. The solution is also difficult to apply in cases where the classification properties are not available for all results. Another considered solution is to apply a directory name that corresponds with the date of storing. The date of storing is a useful indication for administrators and also its value is always known. The date the benchmark was measured was also considered, but this date format differs for each benchmark type. Also on many machines the system date might not be set correctly. Finally, the date of storing the benchmark in the repository was chosen as the benchmark directory name.

Another task of the repository design to deal with is to establish effective searching. The repository must be able to provide a list of benchmarks which satisfy user defined searching criteria. Keeping results in the original format causes the necessity of recurrent processing. To avoid the recurrent processing, the internal data structure should be also established. The internal data structure does not have to copy the whole content of the original format. Searching according to the requested parameters is the most frequently used operation on top of the benchmark storage. As a result, it is necessary to create a searching structure for each

supported property representing the appropriate association of a benchmark result with a configuration description item. At this stage, we can consider using the common data format as the storage for searching structures. This implies that both the original and the common data formats are stored alongside each other. Yet this has proved to be ineffective, because the need of processed data is not so common as the need of properties that hold the configuration description. For each stored benchmark, a list of properties can be stored alongside the results in a separate file. While searching, the data structure of stored results will be ignored, so the searching process is faster. The advantage of the solution is also the possibility of a quick preview of the stored results without the knowledge of their structure. What is also useful for the automated processing is the fixed format of the property storage structure that is shared for all benchmark types.

Decision

The storage is realized using the standard file system without any database. The storage hierarchy is a tree directory structure. In the root directory, there are subdirectories for each supported benchmark type. In the benchmark subdirectory, there are directories for each stored result. The directories are named according to the date of storing. A particular result directory contains the original structure of the stored data and newly added data obtained from result processing. The advantage of the solution is a human browsable format of the repository and no dependency on external software.

In order to replace the database functionality, the repository provides a result indexing support. There are two types of indices - the main and the property indices. The main index provides mapping between a storage path and a unique benchmark identifier. The property indices provide mapping between a value of a property and a benchmark identifier. For each benchmark type, there is always just one main index. The property indices are created for every defined property.

The stored files are parsed to get the included configuration description, and the acquired properties are stored in a separate file alongside the original files. The information necessary for a benchmark identification is also persistently stored in the same file. The file structure is common for all benchmark types, so the repository is prepared for the automated processing while keeping the original format of stored results.

A common data format is generated on demand from result data in the original format. The advantage of the generation on demand is the recency of acquired data according to the current processing rules.

2.3 Parsing

The performance evaluation output differs for each type of benchmarking software. Many of the result formats are proprietary. The repository must provide *customizable support for multiple result formats including plain text and XML formats*.

Details

The repository must be able to process all possible benchmark result formats. There is a need to find a way of parsing that is flexible enough to accept most of the used formats. The

benchmarks usually contain more samples which differ in the benchmark setup. The parsing process should reflect the data structure, preserve the stored information and associations between samples and benchmark setup values. The parsing should be flexible and configurable in order to reflect the changes in the result formats or to provide a possibility of adding a new benchmark type support. Adding the benchmark type support should not cause the necessity to modify the repository source code.

Choices

In order to realize the requirements for parsing, it is necessary to design a universal set of parsing rules. The parsing rules should be also readable for the user and provide the ability to express the detailed structure of benchmark results. In many cases, the token structure must be defined during the parsing, and the tokenizers can change while processing a file. Because of the readable format of the parsing rules and the various structure of parsed results, the usage of context-free grammar was disapproved.

The parsing rules should be general enough to cover all types of benchmark results. There is a lot of ways to specify the set of rules describing the benchmark data and their associated attributes. The more general the description is, the more expensive the data processing and evaluation. So the aim is to find a compromise between the level of generality and speed. While processing the design, a number of description methods using various levels of generality were considered.

The first of three considered solution is geared towards speed. The approach is to design and implement an extensible modular architecture which enables to create new modules or plug-ins for concrete benchmark types. Thus the solution allows fast processing of the known data types and results. During the storage process, the user specifies the type of benchmark, the appropriate module is loaded and the data are processed in the fastest way. The main disadvantage is the high cost of maintenance and the low level of generality. Even a simple modification or an extension of the result format would lead to a modification of the source code. Thus this solution was rejected.

The second considered solution aims for a high level of generality. Elementary benchmark formats are defined as an output of the result formats analysis. Each stored file is assigned to a result class according to its structure. Thus, for each benchmark, there is a general description which determines the result structure and selects the appropriate parser. Now, the repository is able to parse the stored results and get the required information. A part of the benchmark description is a configuration description source location and a data type specification. The specifications are included in property definitions which are configurable without the necessity to modify the source code. The support of various result format classes is realized using the modular architecture, so adding a new result format class support needs to develop a new module. The problem of the solution is the speed. Considering the count of properties M and the count of lines in a file N, the resulting complexity is $M.N$. So the storage and processing of large files with a lot of defined properties is very slow. To optimize the property location determination, for each module, a preprocessing engine should be established, which increases the implementation requirements. This solution was also disapproved, because the file structure is often very complex and many different modules should be created; but the selected concept is a basis for the next consideration.

With respect to the requirements and the previously considered solution, the third solution

is established. The aim is to propose a speed optimization with keeping the high level of generality. The main disadvantage of the previously considered solution is the high cost of determining the property location and a complexity of parsing modules. To eliminate the problem, there is a necessity to determine the structure format and to specify the exact location of each property in the stored results. For the detailed description of the whole result, a *benchmark model* is established. The benchmark model contains a specification of result format classes, which provides the tokenization facilities, a detailed description of the data structure and the exact location and data type of each property occurring in the stored result. Considering the count of properties M and the count of lines in a file N, the resulting complexity is $M+N$. The modification of the benchmark model description does not cause the need to modify the source code. The repository defines a wide range of modules, which reflect the result format classes; so there is a need to add a new module only if the new format class support is added.

Decision

The selected solution supports the highest level of generality. The results are divided according to the data structure, and the format classes are established. For example, the format of text files can be distinguished with the way of token acquirement; the coma, semicolon or white-space character delimiters are supported. For each supported format class there is a corresponding module in the repository, which provides the parsing facilities and the correct data interpretation. There is the benchmark model used for the specification of the results structure.

The benchmark model describes the data structure and assigns the corresponding result format to each stored result. There is a possibility to specify a type of stored file. The file type specification is necessary to choose the correct result tokenizer. The tokenizer is processing the stored files and provides the elementary structures for next parsing. For each elementary structure, it is possible to choose an appropriate parser, which converts the input data into the repository common data format. The model also includes the property definitions. The model description is general enough to support most of the data structures used for benchmark results. The requirements for the model are the support of repetitiveness and nesting of data, and the keeping of the associations between benchmark setup values and a processed sample. Thus the model is defined as a tree structure, and the model processing is directed by the content of stored results.

The repository uses the modular architecture for tokenizers and parsers. The aim of the results analysis and the repository design is to affect the widest possible set of benchmark types, because the creation of a new module needs an implementation in the source code.

The adding of a new benchmark type support is realized by the creation of an appropriate benchmark model. After the benchmark structure is analyzed, the corresponding data format class is found and a detailed result description is created. The processing of a benchmark of the described type is realized by the predefined parsers, which are used according to the model definition.

2.4 Configuration description

The repository should provide *the ability to store configuration description alongside results*.

Details

The configuration description is often part of stored results; however, in many cases, it is not entered correctly or there is a need to extend or specify the stored settings. The values from the configuration description are often used as parameters for searching, so it is possible to store it separately from the results data.

During the results processing there is also repository processing data generated. This data have to be stored alongside the results as well as the configuration description.

Choices

The storage of the configuration description is closely connected with the original form and parsing problem. It is necessary to keep the stored result directory structure, the associations with the benchmark model, the benchmark origin, and the storage date and location. During the storage process, new data is also created. All mentioned items together form *metadata*. The metadata is data used to describe other data. The repository uses metadata to describe the stored results.

There are a lot of ways of implementing metadata. The first of two considered possibilities is to store all metadata together in a separate directory. This concept is convenient for automated processing, but the human browsable format of the repository is lost. The other possibility is to store the metadata alongside the appropriate result. The repository keeps the human browsable format, because in each result directory, the original data are stored together with the metadata. For the user it is easy to get to know about the structure, the storage date or the author of the benchmark. The automated processing is also possible using the global searching indices, which do not need to be stored alongside the results, because their content is overlapping.

Decision

The benchmark description is realized using the metadata. The metadata structure is the same for all benchmarks in the repository, so there is a possibility of automated processing. The metadata consists of the repository processing information, the result origin information and the stored content description. The description covers the associations between the configuration description and the stored samples.

The metadata content is overlapping particularly with the stored result and the indices content. The storage capacity overhead is very low in comparison with the benefits of the metadata, such as the quick overview of a benchmark directory content.

2.5 Evaluation

The repository should provide the facility of the display output generation. The repository should support a set of observations or a part of a sample, a single sample, and a set of samples as a source for the output generation. The processed data are usually very large; there is a need to determine an effective way of graph generation from the benchmark data.

Details

The evaluation process interconnects the storage and the plotting systems. The process evaluates the results in a common data format, which is provided by the storage system, and returns transformed values without the unnecessary content. The set of processed results is very large, thus an incorrect design of evaluation can lead to memory thrashing and system breakdown.

The evaluation process should provide an effective way of describing values, which are specified by the user. Additional processing is needed, such as the aggregation functions calculation. The repository should provide the support for nested function processing.

Choices

The goal of the evaluation is to provide the plotting system with only the values specified by the user. The input is result data in a common data format, which is represented with a tree in memory. Other input is a detailed specification of values for selection. Because of the tree structure of the common data format, it is necessary to specify the tree nodes that should be selected. The tree contains the data only in its leaves, the decision nodes contain the representation of the data associations with the benchmark setup. Thus the path through the correctly created tree covers all important values of the benchmark setup.

The evaluation process imposes high demands on memory usage. For example, the repository should provide a facility to display the minimum of all measured samples in a plot. The processing of one request can include all stored samples of a benchmark type, so it is necessary to evaluate the samples in sequence. The maximum memory usage is equal to the sum of the maximum size of all processed samples and the evaluation output size.

In many cases, it is necessary to process a number of recurrent evaluations on a benchmark. The solution should not lead to repetitive reading of benchmark data and creating an internal data format. The considered solution creates a new output tree structure for each evaluated function. The input data are shared within the processes of evaluation for one benchmark data. The output tree structure groups only the nodes that meet the required conditions. The values in the leaves do not need to be copied, as a consequence, the memory usage will not grow linearly with the number of processed evaluations.

There are many ways of processing the aggregation functions. The first of two considered solution is based on evaluating functions as part of the plot generation process. The interconnection is motivated by the close relationship between the output plot and the processed function. The disadvantage of the interconnection is the impossibility to compose functions. The second considered solution examines the composition of functions. The input and output form of the internal data format is defined for every function. This form is identical for all functions, as well as for the path request processing. Thus there is a possibility to compose functions using the nested functions.

Decision

The position of a value in the tree is described with a *benchmark path*, which consist of a list of regular expressions and expression separators. The evaluation of the benchmark path corresponds to a reduction of the source tree for matching nodes. The benchmark path is closely connected with the benchmark model definition; every element of the benchmark

path corresponds to a decision node, which is defined in the benchmark model. For every selectable value, a variable is defined. Variables can be specified by the user within the plot request. For example, a query can contain a *MessageSize* variable, which determines the message size during an experiment; a plot request with the value of *2048* selects the samples, which were measured with the message size of 2048 bytes.

Definitions of the used aggregation functions are a part of the configuration file as well. The repository provides elementary aggregation functions, such as the minimum, maximum, average or median function. The path evaluation requests and function definition elements form a hierarchical structure, which is called a *query.* For each plot request, a query is predefined. The query definition contains also the plot description and constraints for the input set of benchmarks.

The evaluation of a query is performed on all samples obtained from the storage system. For each sample, an identical process is used. During the evaluation, there is only one sample in memory in order to minimize the probability of memory overflow. The complete query is processed on the sample, and only the output values are stored for future retrieval. The output from all samples forms the input for a plotting component.

2.6 Plotting

The repository should provide *customizable support for multiple display formats including box and whisker plots, density plots, history plots*. The display output should be usable in web browsers and other existing applications. The generation of display output is a slow process, so it is necessary to design and implement speed optimizations.

Details

The repository should provide the facility of the display output generation from all support benchmark types. The resources for a plot can be a single sample or multiple samples from one or more different results. The repository also provides the aggregation functions. The output from these functions applied on multiple samples can be also a source of plotting process.

The other problem solved by the repository is the data type of stored values. The timestamps can be represented as a date-time value or a numeric value representing a multiple of a nanosecond with predefined scale. The output values can be from a specific range, which depends on a type of sample or type of benchmark, so the display output should be scalable for users.

The output calculation and plotting process is very time- and resource- consuming, so it is necessary to optimize it. It is necessary to prevent the repetition of completed tasks and design resource-saving algorithms.

Choices

There is a lot of ways realizing the plotting. The simplest method is to generate a plotting component for every plot type. With respect to the storage analysis, the input of the plotting process is the repository common data format, so the plotting components do not need to

understand the structure of stored results. Using separate components for each plot type causes the repetition of some processes, for example the aggregation functions calculation.

To solve the problem, the separation of the plotting resources calculation and the plot generation is considered. The calculation phase is common for all plot types, so it is possible to group it into a shared component. The output of the calculation is also common for all plotting components. The implementation of plot dependent operations is contained in the plotting components. The plotting components differ in the interpretation of the calculated values and provide the output with requested features.

The plotting process consist of two phases. The first phase aggregates the relevant data interpretation and the output values generation. The second phase includes the display output generation. There is a lot of possibilities to generate the display output. There are bitmap and vector graphic formats, and there is no requirement for choosing one of them. Both formats are integrable in external applications, both have specific advantages and disadvantages. The bitmap formats have a better application support, but there is a possibility of information loss. In order to support multiple graphic formats, a separation of the calculation and the output generation phases is needed. The calculation phase is specific to each plot type, the generation phase depends on the graphic format.

The display output is scalable and configurable, the configuration facility differs for each plot type. The output generation does not need any specific settings, the facility to generate the basic primitives is enough to build any output. Thus it is necessary to specify *plot description* as an interface that provides the detailed output values description to the display output generation module. The plot description is represented with an object structure, which is created by the plot dependent module. This module provides the facilities of the output scalability and the output range configuration. The plot description is also saved in a persistent storage.

The repeated output generation can be reduced using caching. There are possibilities to cache the final plot and the plot description. Both possibilities allow the reduction of repeated calculations. The plot description caching allows the user to generate more plots with the same source, which differs in the configuration of plot ranges or other details. The final plot caching still returns the plot faster, because there is no output generation required.

Decision

The plot request processing is realized in three phases. The query evaluation and values calculation as the first phase is followed by the plot description creation and the display output generation.

The plot description is generated in a plot type dependent module, which provides the correct interpretation of calculated data. The input of the plot description generation is a query, which contains the plot settings and evaluation process output data. Every module uses specific algorithm for proper interpretation of input data. The output of the phase is the plot description object structure that is the same for all plotting modules. The structure is serializable, so it can serve as a source for persistent storage with caching facilities. The structure consists of a plot type specification, a plot axis description and a set o values, which contains the required output values.

The display output is generated independently on the plot type from a plot description. There is a possibility to select appropriate graphic format without the necessity of adding the plot types support. The output of the display generation can be cached in order to achieve

higher speed. The repository does not provide the facility of the final output caching, but it can be implemented in the application server built on the top of the repository.

2.7 Security

The repository should *support various levels of result trust*. The results origin varies, so the repository must provide the facility to distinguish the level of trustworthiness of the stored data.

Details

There are many benchmark results sources, with respect to the benchmark purpose, there is a motivation to publish incorrect results. Because the results origin varies, the repository must provide the facility to distinguish the known and unknown result sources. The identification should be transparent in the maximum possible way. The access restriction in not requested.

Choices

The repository is designed with respect to easy usage, thus the required login would lead to complicated cooperation with the repository. The repository provides the facility to set the benchmark author, the setting is optional.

There is a need of verification of author data. In the network, the basic identification is the network address. Although the repository is designed to work in the network, the repository is conceived as a database application without the possibility to get the user network address. The close cooperation with the application server is needed.

Decision

The repository provides interface for adding the author name, contact and the network identification. The combination of the author name and network identification is the source for the level of trust calculation. For each stored result, the author details and the network identification are stored in the metadata and can be used as parameters for searching or filtering. The level of trust is always calculated, because the benchmark author trustworthiness can change in time.

2.8 Summary

The analysis of the requirements establishes the elementary structure of the repository. The repository should be conceived as a monolithic application with the emphasis on the maximum configurability. The configurability is realized using a robust processed data description, which forms a benchmark model. For each stored benchmark result, a set of persistently stored attributes is also introduced.

3 Repository design

This chapter provides a description of a detailed repository design which is based on the repository requirements. During the analysis, the repository is considered as a monolithic application. In the following paragraphs, a detailed description of the repository storage, evaluation process and plotting generation will be established. The definition of public interfaces and exchange data formats is also mentioned.

3.1 Storage system

The repository provides a configurable persistent storage system that supplies the database functionality. The purpose of the repository does not require a complete implementation of the database functionality. In some cases, it is not even desirable. There is no transaction support; the on-line results storage can be long-running and locking of shared resources a long time is not effective. Moreover, the locking of resources during the result storage leads to the increase in the response time. To avoid collisions, locking of shared resources is implemented for elementary operations.

The storage process is designed with respect to configurability and extensibility; for implementation, a builder design pattern [15] has been chosen. For each supported benchmark type, an individual builder is created on the basis of benchmark model description in the repository configuration. The builder prototype instances are created dynamically during the initialization of the repository and stored in memory.

The insertion of a benchmark causes the retrieval of an appropriate builder prototype according to the type of the inserted benchmark. A new instance of builder is created using the clone method invoked on the retrieved builder prototype. The obtained builder is used for storage. The storage is split into elementary operations; there is a file insertion and a directory insertion operation support. Each inserted element is processed and stored to persistent storage. The metadata is kept in memory until the storage finish method is invoked. During the invocation, the storage of a benchmark result is finished, the metadata is stored in persistent storage and the benchmark result is registered in the repository searching structures.

Persistent storage

The persistent storage is realized using the file system, the results are stored in separate subdirectories distinguished by the benchmark type and storage date. The storage date is considered with a second precision. Collisions are solved using a unique suffix, which is an ordinal number. In each result directory, appropriate metadata is stored alongside the result content in the original form. The indices are stored separately from the result data.

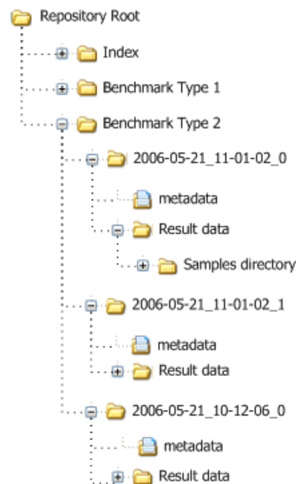


Figure 1: Storage structure

The example is illustrated in the figure 1. In the Repository Root directory, two benchmark types directories and a shared index directory are stored. The Benchmark Type 2 directory contains three results stored in particular directories named after the date of storage. Each result directory contains a file with metadata and a directory structure with measured samples. The samples structure is individual and is recorded in the appropriate metadata.

Benchmark model

During the result storage process, each stored file must be linked with a description in the benchmark model. The file is parsed according to the parsing rules defined in the model, but not converted into the common data format. The aim is to get the included configuration description that is defined by the properties in the benchmark model. The model also defines the way files are assigned to the appropriate sample. The association among the stored file, the benchmark model element definition, and the appropriate sample is persistently stored in the metadata. Thus the future processing does not need to parse the whole content of the stored result, only the necessary files are processed to get a sample. The properties are used for searching the stored results, thus they are inserted into searching indices, and they are also stored in the metadata.

Results identification

While processing the results, there is a need for a unique identification of each stored sample. A single result can contain multiple samples, which differ in benchmark setup or configuration description. To support such a complex result, it is necessary to design a multilevel benchmark sample identifier, which would cover the type of the stored benchmark, the result storage identification and the sample.

The benchmark type is used by the repository at all levels of processing. For each specified benchmark type, there is a separate storage mechanism. The benchmark type is determined

by a string constant, which is defined in benchmark model description. Thus each benchmark identifier contains a benchmark type definition.

The storage process of a single result is an atomic and consistent operation; for each stored result, a unique storage identifier is assigned. The identifier is mapped to the storage path and is called a *benchmark id*. To distinguish more samples in a result, each sample has a sample identifier assigned. The sample identifier is called a *sample id*. In the repository storage system, both identifiers are represented with an ordinal number. In the benchmark model description, there is a string representation for a sample identifier. The string can correspond to a variable, an input element name or a definition. During the result storage process, there is a mapping between the appropriate sample identifiers. After the storage process has finished, both sample identifiers are persistently stored; however, only the ordinal number representation is used in the subsequent processing.

Metadata description

Metadata is the output of the storage phase. Metadata contains information about the result type, result origin and result structure. The result structure information contains a detailed description of samples present in the result.

The repository metadata forms an object structure, which is defined using XML Schema. Thus the object structure is serializable to well defined and standardized XML files. The fixed form of metadata is convenient for automated processing, each metadata can be processed in the same way, irrespective of the original result format. The metadata structure contains a benchmark identifier, a benchmark type, a storage date, a storage path, and benchmark author information. These values are shared for all samples in a result. Within metadata, each sample has its own section assigned. A section is identified by a sample identifier, moreover, it contains a description of the storage structure in the file system and a set of properties that contains the configuration description associated with the appropriate sample. The list of properties available for each benchmark type is specified in the benchmark model description.

Indices

Since a direct access to the stored result is not effective, optimization must be established. Its goal is speeding up searching, which is one of the most frequently performed tasks. This is realized using indices. In the repository storage system, there are the main index and the property index. Both indices are based on a common persistent implementation, only the purpose and the interface differ. The persistent indices are implemented using the serialization of objects.

The main index provides persistent mapping between a storage path and a benchmark identifier. In the repository, there is only one main index for each benchmark type. During the storage process, a unique benchmark identifier is assigned for each newly stored benchmark. The obtained benchmark identifier is stored together with the appropriate storage path to the main index. Thus, having the benchmark identifier, the main index can be used to locate the storage path.

The property indices provide mapping between a value of a property and a benchmark identifier. In the property index, there is a list of sample identifiers referring to the appropriate result for each property. The storage system distinguishes among various samples within a

stored result, so the property index has to refer to the stored sample, which is represented by a sample identifier. Each property index is able to generate a list of stored property values. This list can be used in the graphical user interface as a list of permitted values for searching requests.

Searching

The repository provides a facility for searching within the stored results. Searching is realized through the cooperation with property indices, which implies that the supported parameters for searching are defined in properties. For each property, a list of available values can be obtained to provide the user with the relevant data. The repository supports four types of searching input; there is a *simple*, *set*, *range*, and *regexp* input.

The simple input allows the user to search according to a single value. The set input allows the user to search according to a set of vales. The values must be provided explicitly as an enumeration. The range input allows the user to search within a specified range of values. This kind of searching parameters is supported by the repository only for the property types, which allows their mutual comparison. The regexp search input allows the user to search depending on the match with a regular expression. The supported format of regular expressions is the Java regexp format. This kind of searching parameters is supported only for the string property types.

The searching algorithms are implemented in indices. The searching interface provides only a high-level abstraction, which is usable in other applications and comparison for the users. The result of every searching process is a set of sample identifiers.

3.2 Evaluation

The process of evaluation covers all operations performed on the stored results. There is no write operation during the process. The goal of the evaluation design is a detailed specification of searching, parsing and common data format construction.

The evaluation usually starts with finding samples matching the given criteria. The output of the search is a set of sample identifiers. Having the sample identifier, the storage system is able to locate the storage path, read the metadata description, find the proper sample files, load and parse the stored data.

Use of benchmark model

The storage layer provides a facility to generate a common data format for each sample stored in the repository. For the common data format generation, a benchmark model is used. The benchmark model provides a set of parsing rules. The model processing and result data parsing are done simultaneously, the process is directed by the result content, which is taken from the appropriate metadata. During the benchmark model processing, the property definitions are ignored, because the configuration description is not necessary for the common data format construction. First, the metadata is loaded and the appropriate files are selected according to the required sample content. Then, the selected files are read and parsed according to the parsing rules obtained from the benchmark model.

The output of the process is a common data format represented with a tree structure in memory. The tree leaves contain all observations of the processed sample and the decision nodes contain the associations of the observations with the sample setup values. The generation of the tree can be influenced by the modification of the benchmark model description. Thus, for each benchmark, the generated common data format corresponds to the actual model. The model updates take effect without the necessity of the repeated result import. The output structure is optimized for the query evaluation and output generation.

Benchmark path evaluation

During the evaluation, the benchmark path requests are processed. Samples converted into the common data format are the input of the process. For each sample on the input, the same request is processed. The benchmark path request evaluation consists of traversing a tree from its root to the leaves. In each decision node, the appropriate regular expression from the benchmark path is tested against the node name. If the regular expression does not match the node name, the whole branch including the tested node is removed. Finally, the leaves with result observations are grouped.

The decision nodes structure is omitted, the output observations are inserted into a new output node, which is identified by the sample name. The output node structure is standardized and serves as the input for function evaluation or plot generation.

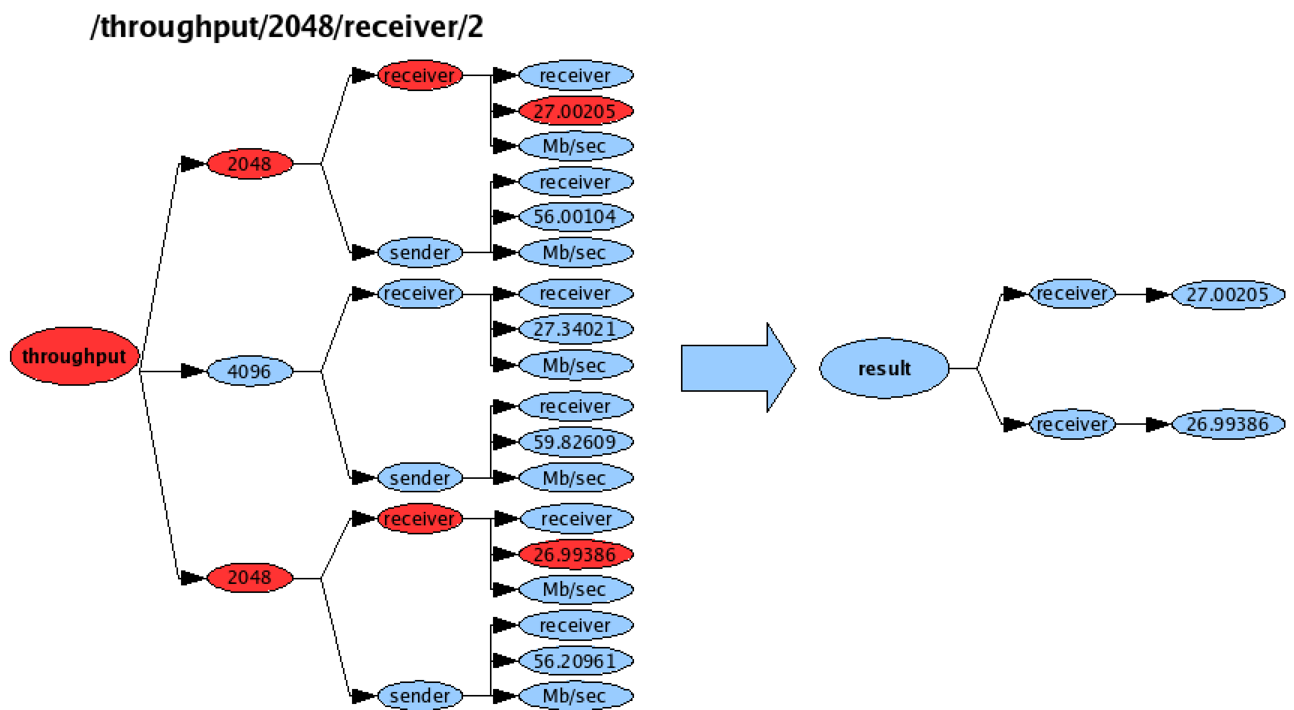


Figure 2: Benchmark path evaluation

Calculation

The calculation process is defined in the benchmark model as a query. For each benchmark type, a set of queries is defined in the model. A query definition consists of a plot resource description and a calculation process description. The calculation process includes benchmark path evaluation tasks, aggregate function calculation task and operation tasks upon the two given sets of tasks. The tasks can be nested, thus it is possible to define a very detailed description of result data processing.

Function	**Description**
Min	calculates minimum of values for each node
Max	calculates maximum of values for each node
Avg	calculates average of values for each node
Med	calculates median of values for each node
Sum	calculates a sum of values for each node
BaW	calculates box and whisker values (min, max, lq, hq, median) for each node
Merge	merges values of all nodes into one node

Table 1: List of Functions

Operation	**Description**
Sub	process a subtraction for all values in child nodes; the structure is preserved
Add	process an addition for all values in child nodes; the structure is preserved

Table 2: List of Operations

3.3 Plotting

The plotting process is realized on the common data format, which is generated during the evaluation process. Plotting is designed with respect to generality, configurability and extensibility. The process is split into two independent phases, the plot description generation and the display output generation phase.

Plot description generation

The plot description generation is plot-type dependent. The plot-type dependency is solved using the builder design pattern [15]. For each supported plot type there is an appropriate builder. The builder functionality is the relevant interpretation of values obtained from the calculation phase.

The output from the generation is a plot description, which is represented by an object structure. This structure is defined in the XML Schema, thus the class instances can be serialized into well defined and standardized XML files. The plot description contains a plot identifier, plot resources identifiers, a plot type, a plot description, a description of both axis, a scale and ranges, and calculated values. After the generation has finished, there is a possibility to modify the scale and the ranges of the plot.

Display generation

The display generation is realized independently from the results and plot types. The input for the display generation is a plot description. The plot description contains only the definitions of plotting primitives. The display generation is realized in plotting components, which provide the plotting primitives interpretation functionality. There are two types of plotting components: internal and external components. The internal components are realized in the repository as modules. The output of the display generation is the plot image, which is integrable in the existing applications, for example, a module for PNG format generation. The external components are realized by postponing the plot generation to a client. In this case, the plotting is realized by a XSL transformation, which creates the SVG format from the input plot description in the XML format. In the repository, no image generation module is implemented. The plot description to image converter is a future extensibility objective.

Plots caching

The process of the result evaluation and plot generation is extremely time-consuming. In many cases, the requests for the plot generation are repeated. In order to avoid the repeated processing, it is possible to use a caching mechanism. Two caching mechanisms are designed in the repository; there is a *plot description cache* and a *display output cache*.

The generation of the plot description is the most time-consuming process. The output from this process is an object structure, which is serializable into XML files. So the plot description phase cache stores the XML files into a given directory. For the identification of stored files, hashing is used. A hash string is calculated from a list of used results and a plot request specification. The hash string is a part of the file name. Hash collisions are not solved with an algorithm; the colliding files are overwritten.

The processing of an incoming plotting request begins with a hash string calculation. From the calculated hash string, a file name is generated. If the file already exists, it is deserialized to an object structure and the calculation parameters are compared. If the comparison fails, the plot is calculated again and the generated file is cached instead of the previously stored file. If the comparison succeeds, the loaded object structure is used for the next processing.

The display output caching uses the same caching mechanism and file naming convention as the plot description caching. This mechanism is designed, but not implemented in the repository, because the display generation is not so time-consuming and the generated files are expected to be larger than the plot description cached files. So the existence of the second phase caching is not critical.

4 Benchmark model

The benchmark model is used for the description of the result data structure, the transformation from the original format into a common data format, the configuration description and plotting requests related to the appropriate benchmark type. The benchmark model consists of a description and an object structure, which corresponds to the description element structure. The benchmark model object structures provide the facility to interconnect the defined processing, parsing and plotting rules with other repository components. The benchmark model description is realized using the XML configuration file. The XML format was chosen because of the flexibility and extensibility.

The benchmark model consists of three types of elements: content elements, description elements and query elements. A benchmark model example is mentioned in the Appendix C.

4.1 Content elements

The goal of the benchmark content is to define a fixed data structure, which is able to describe both the known structural types and the types where the structure is not implicitly given. The content elements can be nested in themselves, so they are formed into a hierarchical structure. The base of the structure is formed with the directory and file elements. The file elements contain other elements according to the type of described data and the file structure. For text files, the following content elements are defined: line, sequence, choice, property, and variable. For XML files, the following content elements are used: xmlelement, xmltext, property and variable.

Directory element

The *directory* element is used to describe a stored directory. Although the directory does not contain any parse-able data yet, it can be an important part of storage process; the directory name can be used as an association with a benchmark setup value. The name of the directory also distinguishes its content to the appropriate sample.

Attribute	Description	Allowed values	Required
id	directory identification string		yes
bid	sample identification	dirname, constant, variable	no
bname	sample identification		no

Table 3: Directory attributes

File element

The *file* element is used to describe a stored file. The *type* attribute specifies the appropriate reader used to read the file content. The file element should be present as a child element of a directory element.

Attribute	Description	Allowed values	Required
id	file identification string		yes
type	type of file	text, xml, html, log	yes

Table 4: File attributes

Line element

The *line* element is used to describe a line of a text file. The *type* attribute determines the parser type for the line content parser processing. It is the only element able to parse the text file input data. The parsing is described in the attributes. The line elements also build a simple tree which is provided to parent elements. Each line can be a part of one or more samples. The *node* attribute specifies, wheather create a node of a common data format from a particular element. The *true* value means the node is created, the *false* value means the node is not created, but the node content is placed into the parent node. The *none* value means the node content is dropped.

Attribute	Description	Allowed values	Required
type	type of line format	simple, config, separator, bracket	yes
separator	separator, only for separator type		no
datatype	data type of line content		no
source	source of data type definition		yes
node	node creation flag	true, false, none	no
nodename	node name definition		no
nametype	type of node name source		no

Table 5: Line attributes

Sequence element

The *sequence* element is used to describe an iteration of its child elements. The attributes specify the count of repetitions and the source of this information. The nesting of sequence elements can be used for specifying a tree structure.

Attribute	Description	Allowed values	Required
type	type of count definition	word, number, variable	yes
count	count definition		yes
node	node creation flag	true, false, none	no
nodename	node name definition		no
nametype	type of node name source		no

Table 6: Sequence attributes

Choice element

The *choice* element is used to describe an alternative of its child elements. The alternative consists in the sequential evaluation of the child elements; the first matching element is used as a result of the alternative. Child elements of the choice element can be only line elements.

Attribute	Description	Allowed values	Required
node	node creation flag	true, false, none	no
nodename	node name definition		no
nametype	type of node name source		no

Table 7: Choice attributes

XmlElement element

The *xmlelement* element is used to describe the structure of XML files; it represents an element of the XML document, the name of the element is used for identification. Xmlelements can be nested, their structure reflects the structure of XML file. The unnecessary elements for processing need not be present in the benchmark model. Child elements of the xmlelement element can be xmlelement and xmltext elements.

Attribute	Description	Allowed values	Required
name	name of XML element		yes
datatype	data type of XML element content		no
source	source of data type definition		no
node	node creation flag	true, false, none	no
nodename	node name definition		no
nametype	type of node name source		no

Table 8: XmlElement attributes

XmlText element

The *xmltext* element is used to describe the structure of XML files; it represents an inner text of a XML element. The inner text is processed using the text reader and the appropriate parser specified with the *type* attribute.

Attribute	Description	Allowed values	Required
type	type of inner in XML element	simple, config, separator, bracket	yes
separator	separator, only for separator type		no
datatype	data type of XML text content		no
node	node creation flag	true, false, none	no
nodename	node name definition		no
nametype	type of node name source		no

Table 9: XmlText attributes

Property element

The *property* element is used for locating the configuration description values. The property elements are used only during the insertion of files, when the metadata is created. The properties forms the source for indices.

The property elements cannot be nested. The only relevant parent elements are the line element and xmlelement element.

Attribute	Description	Allowed values	Required
name	name of property		yes
field	source definition of property		yes
type	data type of property		yes

Table 10: Property attributes

Variable element

The *variable* element is used for description of values, which are necessary to determine the result structure. The list of variables is stored in a parsing context, the value of a variable is used when the appropriate variable is referenced from other content element.

The variable elements cannot be nested. The only relevant parent element are the line element and xmlelement element.

Attribute	Description	Allowed values	Required
name	name of variable		yes
field	source definition of variable		yes
type	data type of variable		yes

Table 11: Variable attributes

4.2 Description elements

The *description* section is used for description of repository features that are not related to the content of the benchmark. The description elements cannot be nested. Currently the search subsection is available.

Search element

The *search* section is used for description of searching parameters. The section cannot be nested and contains a list of the index elements.

Index element

The *index* element is used for definition of used indices. For each element, the appropriate index is defined. Only the defined indices are created. Index definition is related to a property, the appropriate property is specified with the *property* attribute. The property need not be defined in the content section, the concrete property can be inserted directly.

Attribute	Description	Allowed values	Required
name	name of index		yes
property	name of property		yes

Table 12: Index attributes

4.3 Evaluation elements

The *evaluation* section is used for definition of request templates for plotting. The evaluation section contains a list of *query* elements.

Query element

The *query* element is used for definition of a plotting request template, which consists of a data evaluation definition and description of searching parameters. Query element contain a *xasix* element, a *yaxis* element and a root of evaluation elements hierarchy. The element cannot be nested.

Attribute	Description	Allowed values	Required
name	name of plotting request		yes
graph	type of graph	simple, histogram, density, baw	yes

Table 13: Query attributes

Xaxis and Yaxis element

The *xaxis* and *yaxis* elements are used for definition of a plotting axis properties. The elements cannot be nested and cannot contain child elements.

Attribute	Description	Allowed values	Required
name	name of axis		no
type	data type of axis		no
unit	unit on the axis		no
min	minimum value		no
max	maximum value		no

Table 14: Function attributes

Function element

The *function* element is used for definition of a function calculation. The function element can be nested, and can contain one child element of the following type: a function, a operator or a path element.

Attribute	Description	Allowed values	Required
type	type of function		yes

Table 14: Function attributes

Operator element

The *operator* element is used for definition of an operator calculation. The operator element can be nested, and can contain two child elements of the following type: a function, an operator and a path element. The order of the child elements matters if the operator is not commutative.

Attribute	Description	Allowed values	Required
type	type of operator		yes

Table 15: Operator attributes

Path element

The *path* element is used for definition of a benchmark path expression on the top of a common data format. The path element cannot be nested, and do not contain any child element.

Attribute	Description	Allowed values	Required
value	benchmark path definition		yes

Table 16: Path attributes

5 Implementation details

In this section, the architecture of the repository is described. The implementation details on the level of object model are established. The future extension possibilities are also mentioned.

5.1 Repository architecture

The repository is designed using the *three-tier* model in which the client interface, application logic and data storage and access are developed as independent modules. The advantage of three-tier model is possibility of running modules on separate platforms.

The repository uses web services, thus the published interface is platform-independent. Thus there is a possibility to create various multi-platform client applications.

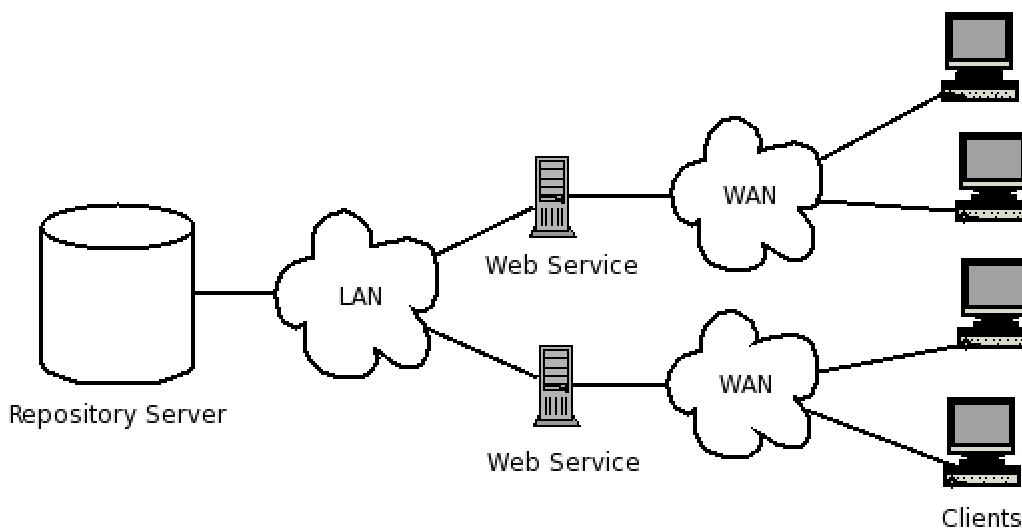


Figure 3: Deployment diagram

5.1.1 Data storage tier

The data storage tier forms the most important part of the repository. The tier is conceived as a server application, which uses the Java RMI technology to provide its functionality. The server is represented with a class, which is designed with the usage of the facade design pattern [15] in order to simplify the interaction with the repository.

The server is assembled from modules which correspond to the internal components. The user is separated from these components using the facade. Each internal component is defined as an interface. Concrete objects implementing the interfaces are obtained using the factory method and abstract factory patterns [15]. For each module, there is a proper factory class.

5.1.2 Application tier

The application tier is realized as a web service and provides the platform independent interface. The repository web service does not contain any application logic, all requests are propagated to the underlying repository server. For each request, a new instance of the Java RMI client is created. The request processing is stateless and satisfies the SOA requirements.

There is an exception in the storing of a benchmark result. For each stored benchmark result, the repository server creates a builder, which is referenced with a numeric identifier. It is necessary to supply the identifier for the insertion of all elements that form the benchmark. This identifier is held in client context and does not influence the web service functionality.

5.1.3 Client tier

The repository provide a console utility for importing the results into the repository. The importing utility is conceived as a web service client. All parameters are taken from the command line, and the benchmark result insertion is performed automatically.

The repository can be easily extended for other clients. There is a possibility to build the dynamic web site using the web services.

5.2 Technology overview

All modules of the repository are realized in Java. The implementation requires the Java 2 SE platform with Java Web Services Development Pack (JWSDP) 2.0 or the Java EE 5 platform. For XML parsing, it is necessary to supply a SAX parser module, such as the Xerxes parser.

For building of the repository, Apache Ant is used. For web services, the Sun Java System Application Server from the Sun Java EE 5 package is needed.

For data exchange and storage, the XML data format is chosen. This was done especially due to the ability to store and restore an object structure and native tree hierarchy support. Next, the existence of many quality tools and libraries is of equal importance. In the repository, XML is used for the representation of metadata and plot description. For both, the easy serialization and deserialization is needed, because they are created and used in memory as object structures and stored in the persistent storage as XML files. For these purposes, the Java Architecture for XML Binding (JAXB) is used. JAXB is a part of the JWSDP, and provides standardized interface for serialization and deserialization of XML files to object structures in memory. The XML files must be defined using XML Schema. The description of the metadata format and the plot description format in XML Schema allows wide range of extensibility in future.

For benchmark model description, the XML format is also used. The processing is done using the Simple API for XML (SAX). The SAX parsing is chosen because the benchmark model processing is connected with many actions during the repository initialization and the benchmark model description is only read.

5.3 Object model

In this section, the detailed design including class diagrams is outlined. The object model serves as an introduction to implementation details. The detailed description of the repository

source code is present in the generated programming documentation on the CD-ROM. The repository consists of three parts; the web services and the console client are not mentioned, only the repository server is described.

5.3.1 Repository Server

The repository server is implemented in Java, thus for each component, there is a separate package. The main packages are the *main*, *storage*, *search*, *query*, *graph*, and *util* packages.

The *main* package contains the *NetworkRepository* interface, which provides the wrapper for the RMI server. The class contains an instance of the *RepositoryManager* class, which is designed using the facade design pattern [15]. This package also contains the *ReposException* exception, which is thrown if an error occurs during the repository processing.

The *storage* package contains an implementation of parsing, benchmark model processing and storage system. For benchmark insertion, the *BenchmarkBuilder* class is used. An instance of the class is obtained using the prototype design pattern. The appropriate builder is found, and an initialized copy is created.

The BenchmarkBuilder prototypes are created on the basis of the benchmark model description and are held in the *BuilderCreator* class instance. For each benchmark defined in the model description, a BenchmarkBuilder instance is created. The instances differ in the content description. The BuilderCreator uses the SAX parser. The product of building process is the *BenchmarkInfo* class, which encapsulates the metadata object structure created using the JAXB library.

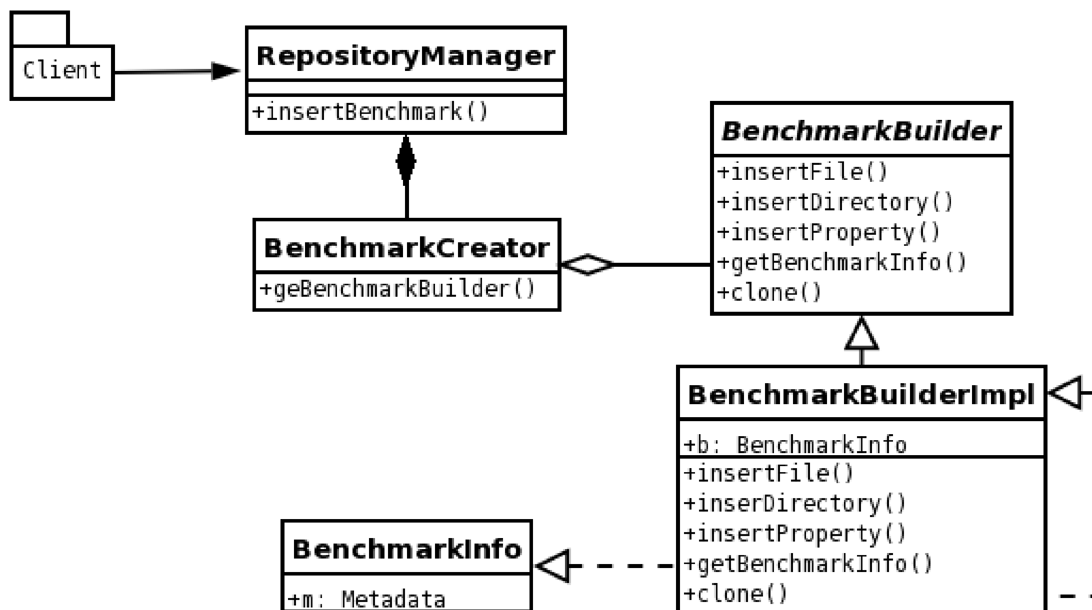


Figure 4: Benchmark builder

The parsing of stored files is defined in the *BenchmarkParser* interface. The *BenchmarkParserXmlImpl* class and *BenchmarkParserTextImpl* implement the BenchmarkParser interface for XML and text files. The *BenchmarkManager* class contains a support for the persistent operation for benchmark of an identical type. For each benchmark type, there is just one instance of this class in the RepositoryManager class. Each stored sample is represented with a *Benchmark* class. The Benchmark class provides access to a sample in the common data format.

Parsing is closely connected with the benchmark model definition, which is contained in the *storage.model* package. For each model element, there is a class with a corresponding name. The *storage.util* package contains a definition of unique identifiers and an internal tree, which is used for a common data format representation. The *BenchmarkId*, *SampleId* and *DataNode* classes are also contained in the package. The input results readers and low-level parsers are placed in the *storage.reader* package.

The *search* package provides the interface for searching facilities upon the data stored in the repository. Actual searching is implemented in the storage system and indices. The searching algorithms are implemented in the *SearchProcessor* class. The searching parameters are exchanged using the *SimpleSearchParameter*, *SetSearchParameter*, *RangeSearchParameter*, and *RegexpSearchParameter* classes. The searching input is stored in the *SearchRequest* class. The output of searching is realized with the *SearchResponse* class.

The *query* package provides evaluation facilities and representation of plot requests of the benchmark model. Each plot request is represented with a *Query* class. A Query class contains the object structure defined with the *IQueryNode* interface. The interface is implemented in the *PathNode*, *FunctionNode* and *OperatorNode* classes. The appropriate instance is created using the *QueryNodeFactory* class. The calculation algorithms are implemented in the *NodeFunction* and the *NodeOperator* classes. For operations upon queries, the *QueryProcessor* class is implemented.

The *graph* package implements plotting facilities of the repository. The plotting process is designed using the builder pattern. The *GraphDirector* class is used as a building director, which contains the building algorithm definition. The builder is provided by the *GraphBuilder* interface. There is an implementation for each supported plot type. The product of building is the *Graph* class. This class encapsulates the plot description object structure, which is easily serializable into a XML file.

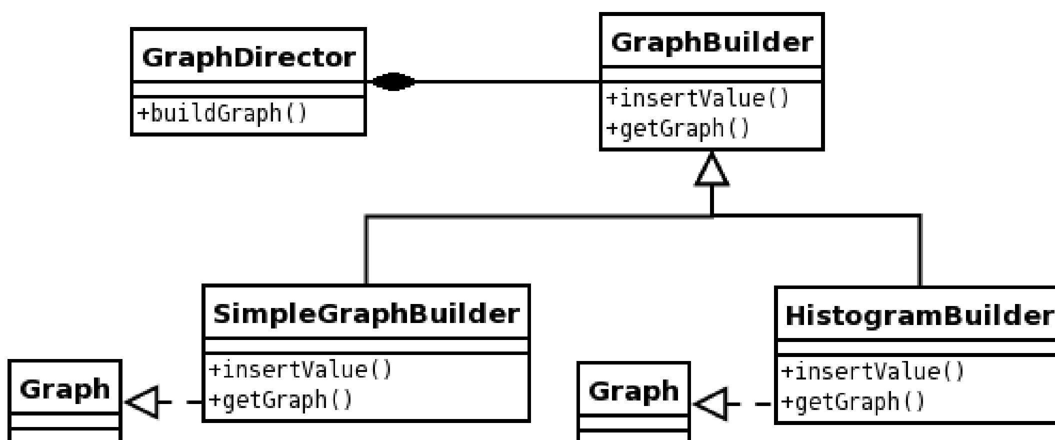


Figure 5: Graph builder

The *util* package implements configuration, logging and indexing facilities. The configuration settings are provided using the *ConfigManager* class. The logging facilities are provided with the *Logger* class. The indexing is available using the *Index* interface. The repository indices are build using this interface in the *BenchmarkIndex* and *SearchIndex* classes. An instance of an appropriate index can be acquired with the *IndexFactory* class.

The *util* package also contains the *util.data* subpackage, which provides the data types hierarchy. The implementation is based on the Abstract Factory design pattern [15]. For each data type, a *DataType* factory class is established. The factory provides the *parse()* method, which return a *Data* class instance as a product. The input for the *parse* method is a string value representing an observation. The appropriate factory is created in *DataTypeFactory* class according to data type definition in the benchmark model for particular observation.

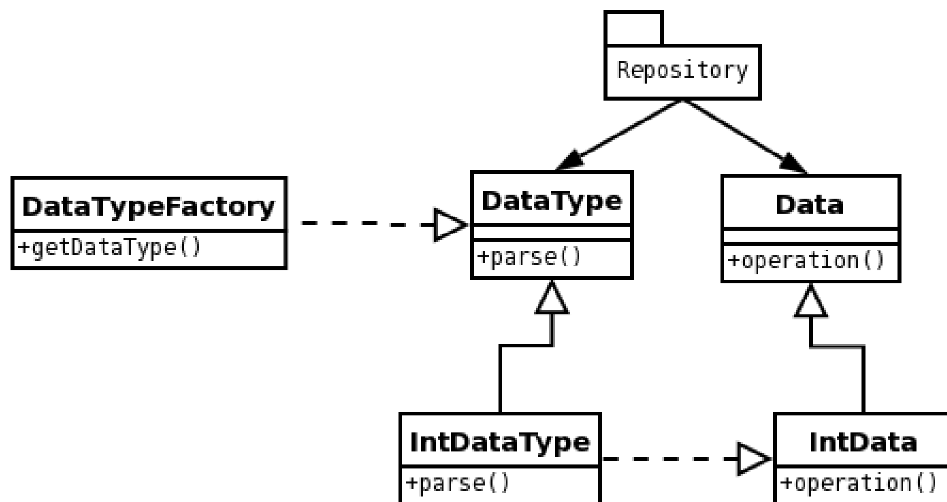


Figure 7: DataType abstract factory

5.4 Extending repository

The repository is designed as a modular system, so it is possible to add a new functionality or to extend the existing interface implementations. There are the abstract factory and the factory method design patterns used, so the adding of a extension means the creating of an interface implementation.

Parsing extensions

The parsing refers to two possible problems. The first problem is a new result class. In this case, a new reader and appropriate parsers must be established. The current version of Network Repository supports both content driven parsing, represented with XML files processing, and benchmark model driven parsing, represented with text files processing. The newly added result class support may lead to creating a complete reading and parsing facilities.

The second problem is a new type of results appears. To overcome this problem, the existing parsing facilities can be used. But the benchmark model description can rapidly grow because of properties repeated specification. Other possibility to solve the problem is a new implementation of a file reader or a line parser. The text file reader is specified in the *ITextReader* interface and is used for reading text files and for generating text lines, which are used as an input for line parsers. A new instance of this interface is created with the *TextReaderFactory* class. While processing a file, the factory method is called according to the benchmark model definition. The line parser is specified in the *ISourceParser* interface. The purpose of this interface is to tokenize a text line and return an instance of the *ParsedLine* class, which is used as a basis for a result processing and a common data format building. A new instance of the interface can be created with the *TextReaderFactory* class during the file processing in the *SourceLine* class.

The using of a newly created reader or line parser can be invoked with the specification of the appropriate reader or parser in the benchmark model description file. The repository must be restarted to reflect the changes in the benchmark model description.

Indexing extensions

In the repository, there are the low level and application level indices implementation. Both types of indices are created with the *IndexFactory* class, which is conceived as an abstract factory design pattern. The purpose of the application level indices is to provide a logic layer upon the low level indices, which are related to storage system. The application level indices are defined in the *BenchmarkIndex* and *SearchIndex* interfaces. The modification of these classes influences the functionality of the repository.

The low level index is defined in the *Index* interface. The particular implementation of this interface is used with both application level indices. The interface provides the persistent index with support of multiple values for one key. The low level indices can be replaced with a more powerful implementation. The problem of the replacement is modification of the persistent storage format. This will lead to a failure in reading of previously created files, so the of re-indexation of stored data must be performed. In the repository, a migration utility is designed. The migration utility reads the content of a repository and loads metadata stored in the root directory for each stored result. The obtained acquired values are indexed in the newly created indices.

Plotting extensions

The plotting system is designed using the builder design pattern. The plotting output is the same plot description structure for all plots, but its construction differs. The difference is realized with the particular builder. The implementation of a builder is an extension of the

GraphBuilder abstract class, which provides the facility of persistent storage and general settings. The *insertValue* methods must be reimplemented. The builder is created using the *GraphFactory* abstract factory, so the factory implementation must be extended to make the new builder accessible.

It is also possible to modify the output format of the plotting phase. The modification of the plotting description is done with editing the *Graph* XML Schema file and regeneration of the code. The changes must be included in the *Graph* and *GraphBuilder* classes in order to keep the relevant cooperation with the rest of the repository. Adding of a new attribute or a new tag is a backward compatible operation.

6 Related works

The concept of Network Repository as a general storage for various benchmark result types is rather unique. During the research, no software project with identical goals was found. Usually, the benchmark vendors provide a set of utilities together with the testing suite. The utilities are able to parse and process the measured data from its suite and generate visualizations. These utilities are not usable for results of different benchmark types, moreover, these utilities are often proprietary. In connection with the regression benchmarking[2], many on-line browsable repositories are established. These repositories usually monitor a concrete software product and the published results are read-only.

BEEN

BEEN[11] is partially a similar project. The goal of this project is to provide automatizing all steps of benchmark experiment from software building and deployment through measurement and load monitoring to evaluations of results. What is similar to Network Repository are the separation of the measurement process from the evaluation and visualization. A difference is that the BEEN project covers all steps of benchmark experiments for a particular benchmark type, using an internal data format for storing results, while Network Repository covers only storage and evaluation, taking into consideration the generality and preserving the original data format. The implementation is currently in beta stage.

CLIF

The CLIF [12]project is a framework dedicated to performance testing. The primary objective of the project is deploying, controlling the execution of, and monitoring of performance tests. CLIF uses the Fractal component model. The measured values are recorded into internal data storage. The data storage includes monitoring of response times, throughput, error rate or computing resource consumptions. Analysis and visualization tools are also provided. The storage is only secondary objective of the project. In comparison with Network Repository, the CLIF project is more comprehensive and the goals of both projects are different.

CCPsuite

The CCPsuite [13] project provides a comprehensive benchmarking platform, which consists of five separate projects complementing each other. The platform covers CORBA benchmarking suites, plotting utilities and a benchmark result browser. The suite does not contain a managed storage; the results from benchmarking suites are stored on file system and serve as a source for the plotting utilities and the result browser. The Network Repository concentrates for storage as a main goal, which makes a difference with the CCPsuite objective.

SPEC

The SPEC [14]is a non-profit corporation, which maintain a set of relevant benchmarks. The measured results are published in a public on-line repository. The concept of searching and on-line browsing is very similar to Network Repository, but the SPEC repository supports only the native corporation benchmarks. The on-line repository fits to the set of supported

benchmarks. The implementation details are not published, thus the design of its storage system is not discussed.

Rubis

The Rubis [9] project is an example of a benchmarking software, which have own repository for its result. Rubis is an auction site prototype and is used for evaluation of the application server performance. The results are generated in a format, which is accepted in the repository without the necessity of futher processing. The repository is on-line browsable, but do not allow to generate the user specified reports. The specialization for the particular benchmark type provides better interconnections among measured results than Network Repository.

7 Conclusion

The assignment of the master thesis covers all operations with benchmark results data, from the data storage to the plot generation. Based on the complexity of the assignment, the web interface was not implemented in order to treat a narrowed topic in depth rather to deal with a complex task without examining the requested details. The limits to the assignment received approval from the thesis supervisor.

During the analysis, the universal, flexible and extensible storage for various benchmark result types was considered to be the most important part of the thesis. The variety of benchmark result types only underlines the importance of creating a robust benchmark model, which forms the basis for processing. The configuration of the model is realized with a benchmark model description that is represented with a XML file.

As part of the realization a benchmark model description, covering the assigned benchmark formats [5-10] was established. The example configuration is mentioned in the Appendix C. The benchmark model description can be easily extended with adding a support for a new benchmark type according to the benchmark model section of this thesis.

The storage implementation uses metadata. The metadata provides a description of a particular benchmark result which is stored in a XML file. Its structure is defined with a *Metadata* XML Schema, which is visualized in the Appendix A. The standardization of the metadata description allows future automatized processing performed on the existing repository storage structure.

The plotting implementation is realized using a plotting description, which contains the output from evaluation of stored results. The plotting description serves as a data source for image generation and is realized as a XML file, which is defined with XML Schema. The *Graph* XML Schema is visualized in the Appendix B.

The repository is realized as a server application, written in Java. The public interface for communication is realized via web services. The web services implementation provides all functionality of the repository, so many platform independent clients can be established. Thus Network Repository is prepared to store benchmark results, which were measured on various platforms. Network Repository can be also used as a resource for an on-line browsable web repository.

References

[1] Kalibera T., Bulej L., Tůma P.: *Generic Environment for Full Automation Of Benchmarking*, in proceedings of Net.ObjectDays 2004

[2] Bulej L., Kalibera T., Tůma P.: *Regression Benchmarking with Simple Middleware Benchmarks*, in proceedings of IPCCC 2004

[3] Bulej L., Kalibera T., Tůma P.: *Repeated Results Analysis for Middleware Regression Benchmarking*, accepted for publication in Performance Evaluation: An International Journal

[4] Tůma P., Buble A.: *Open CORBA Benchmarking*, in proceedings of SPECTS 2001

[5] Sampler Performance Evaluation Tool, *http://nenya.ms.mff.cuni.cz/mbench*

[6] Xampler Performance Evaluation Tool, *http://nenya.ms.mff.cuni.cz/mbench*

[7] ECperf Performance Evaluation Tool, *http://java.sun.com/j2ee/ecperf*

[8] OVM Performance Evaluation, *http://ovmj.org/bench*

[9] RUBiS Performance Evaluation Tool, *http://rubis.objectweb.org*

[10] TAO Performance Evaluation, *http://www.dre.vanderbilt.edu/stats/performance.shtml*

[11] BEEN - Benchmarking Environment, *http://dsrg.mff.cuni.cz/been/project/*

[12] CLIF - The CLIF Project, *http://clif.objectweb.org/*

[13] CCPsuite, *http://dsrg.mff.cuni.cz/~ceres/prj/CCPsuite/*

[14] SPEC - Standard Performance Evaluation Corporation, *http://www.spec.org/*

[15] Gamma E., Helm R., Johnson R., Vlissides J.: *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison-Wesley, 1995

Appendices

Appendix A. Metadata XML Schema

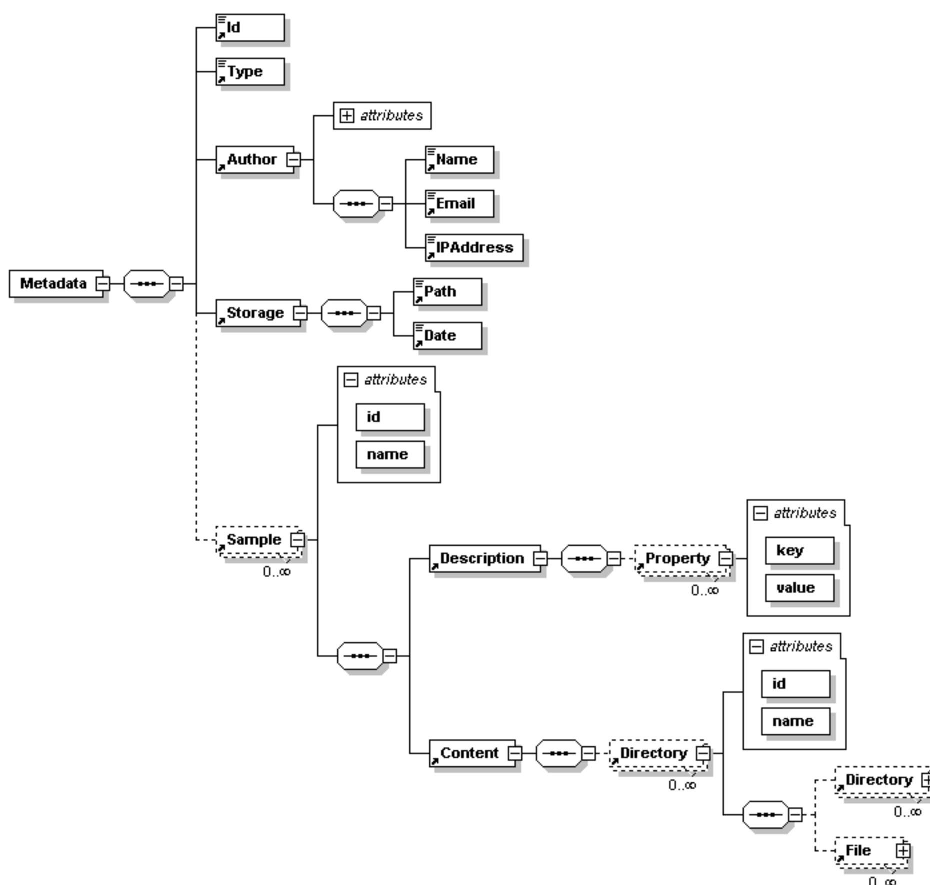


Figure A: Metadata XML Schema visualization

Appendix B. Graph XML Schema

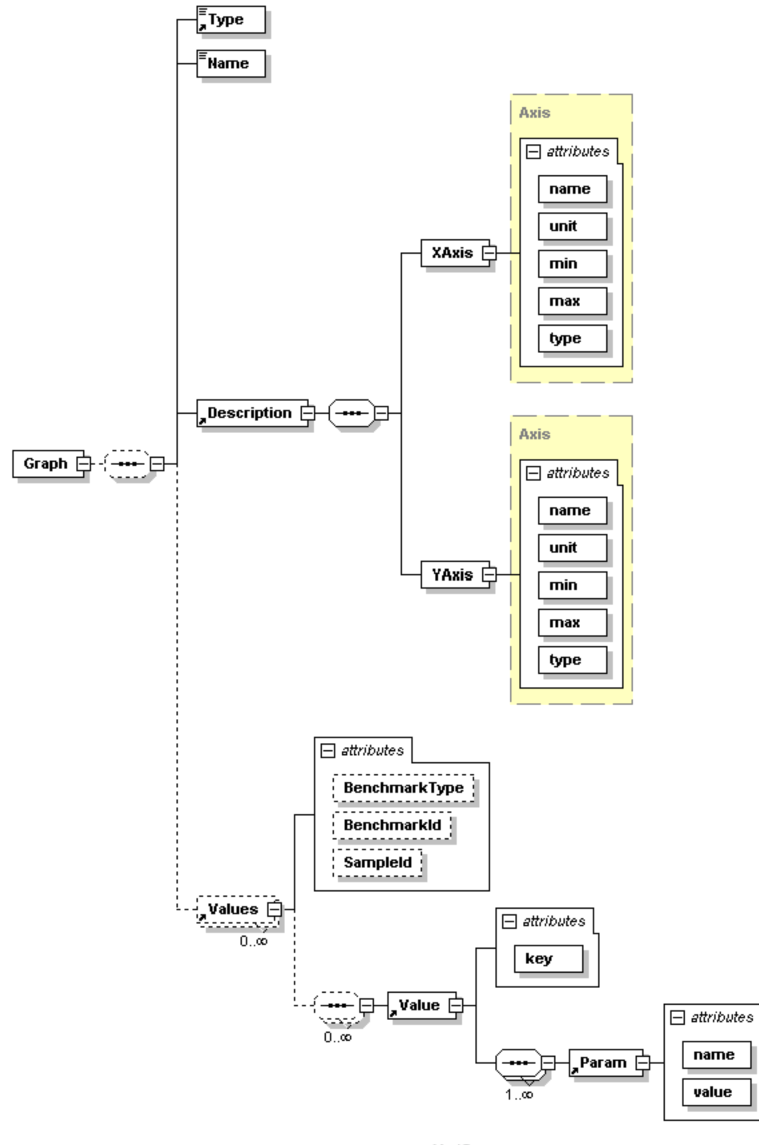


Figure B: Graph XML Schema visualization

Appendix C. Example TAO configuration

```
<benchmark name="TAO">
 <content>
  <directory id="logs" bid="dirname" >
   <directory id="log" >
    <file id="log" type="text" >
     <sequence type="word" count="unbounded" >
      <choice >
       <line type="colon" >
        <property name="Count" field="count" type="int" />
       </line>
       <line type="simple" />
      </choice>
     </sequence>
    </file>
   </directory>
  </directory>
  <directory id="stats" >
   <file id="stat" type="text" node="true">
    <sequence type="word" count="unbounded" node="true" nodename="StatName" name-
type="variable" >
     <line type="skip" node="false" />
     <line type="simple" >
      <variable name="StatName" field="1" type="string" />
     </line>
     <sequence type="number" count="4" >
      <line type="config" />
     </sequence>
     <line type="simple" />
    </sequence>
   </file>
  </directory>
  <directory id="data">
   <file id="throughput" type="text" node="false">
    <line type="simple" node="none" />
    <sequence type="number" count="6" node="true" >
     <line type="simple" node="none">
      <variable name="MessageSize" field="3" type="string" />
     </line>
      <sequence type="number" count="4" node="true" nodename="MessageSize" name-
type="variable" >
      <line type="simple" node="true" datatype="float" />
     </sequence>
    </sequence>
    <line type="skip" node="none" />
```

```
      </file>
    </directory>
  </content>
  <description>
    <search>
      <index name="Count" property="Count" />
    </search>
  </description>
  <queryset>
    <query name="Minimum_Troughput" graph="Simple" >
      <xaxis name=”Sample” />
      <yaxis name=”Minimum Throughput” unit=”MBps” />
      <function type="Min" >
        <path value="throughput/${MessageSize}/.*/2" />
      </function>
    </query>
  </queryset>
</benchmark>
```

Appendix D. CD-ROM content

List of directories:

doc electronic documentation including administration guide

bin binary distribution

src source code

conf benchmark model description for assigned benchmark types [5-10]

xsd XML Schema specification

javadoc programming documentation

data sample testing data