Institut für Parallele und Verteilte Systeme

Abteilung Anwendersoftware

Universität Stuttgart
Universitätsstraße 38
D-70569 Stuttgart

Diplomarbeit Nr. 3243

Development and Evaluation of a Framework for Semantic Validation of Performance Metrics for the IBM InfoSphere Optim Performance Manager

Moritz Semler

Studiengang:	Informatik
Prüfer:	PD Dr. Holger Schwarz
Betreuer:	Dr. Ingo Hotz
begonnen am:	01.09.2011
beendet am:	27.03.2012
CR-Klassifikation:	D.2.4

Blocking Notice / Sperrvermerk

Structure

List of Abbreviations

CTL	Computation Tree Logic
LTL	Linear Temporal Logic
ms	milliseconds
OLAP	Online Analytical Processing
OPM	Optim Performance Manager
SQL	Structured Query Language
SUT	System Under Test
XML	Extensible Markup Language

List of Figures

List of Tables

Development and Evaluation of a Framework for Semantic Validation of Performance Metrics for the IBM InfoSphere[1] Optim[2] Performance Manager

Abstract. Validation is an important field in the software development process. It helps to increase the software quality but is also very expensive and time consuming. To decrease the costs approaches to automate the validation process are necessary. In this thesis a framework is developed, which does not need user interaction to validate the IBM InfoSphere Optim Performance Manager semantically. It is able to validate values of different behavioral patterns. It covers deterministic, semi-deterministic and non-deterministic behavior. The thesis describes the process of the development of the framework. It introduces available approaches and examines them with regard to the suitability for the framework. The found solution is described in theory and a prototype is implemented to apply the solution to praxis. This prototype is evaluated on the latest version of the IBM InfoSphere Optim Performance Manager.

[1] Trademarks of IBM in USA and/or other countries
[2] Trademarks of IBM in USA and/or other countries

1 Introduction

Since software systems become more and more complex, quality is one of the fundamental decision criterions for customers. Consequently the effort and the importance of software validation have increased over the years. Studies report that about 50 to 60 percent of software development cost is spent in the area of validation [1].

Besides the complexity of software, the nature of functions or values to be verified determines the difficulty of the validation process. Functions which compute foreseeable results are much easier to validate than functions computing values which cannot be predicted. So the effort of software validation depends not only on the size of software but also on the characteristics of its functions.

The goal of this Diploma thesis is to develop and evaluate a Semantic Validation Framework for the IBM InfoSphere Optim Performance Manager (OPM), a performance analysis tool for DB2[3] databases. OPM supports database administrators in identifying, solving and preventing database performance issues. To this end it collects performance metrics of monitored database systems and displays them to the user via a web interface and offers the possibility to alert the user about up-coming problems. In this case, performance metrics include all kinds of data helping to make assertions about the health status of the database. For example, how many statements are processed, how often the requested data is located in the bufferpool or which statements caused a deadlock. Further these metrics are values of the underlying system for example the size of the memory or values of the connected clients as IP-addresses or types of used applications. This means, OPM provides performance metrics about the database, the system on which the database is located, and about the clients working on the database.

The Semantic Validation Framework aims to provide an automatic correctness proving for all OPM metrics. This means that it makes assertions about the correct implementation of metrics without user interaction. This process differs depending on

[3] Trademarks of IBM in USA and/or other countries

the nature of a metric. First steps of the thesis are to sort these metrics into categories according to the behavior of their values. This classification is a part of the framework and is used there for deriving correctness criteria to decide whether a metric is implemented correctly. Depending on the category there are different approaches for the validation.

In the thesis, it is investigated if there are suitable approaches and correctness criteria available, which can be used for proving semantic correctness of the OPM performance metrics. The results of this investigation reveal that there are such approaches, fitting for the project and that these approaches have to be adapted to match the special requirements of the OPM environment.

A prototype of the Semantic Validation Framework is developed and integrated into the existing test environment. This environment has been used so far to check the integrity of metrics shown in the web interface of OPM and is extended with the Semantic Validation Framework. It is also able to use an IBM internal tool to run workloads on databases.

The document is structured as follows:

- Chapter 2 provides a detailed insight into the OPM environment. This includes OPM itself, the test environment, and a workload tool. It is pointed out which components are already available and which have to be developed for semantic validation. Reasons and the motivation for the development of the Semantic Validation Framework are shown. Additionally, possible problems for semantic validation are described.
- Chapter 3 begins with a definition for semantic validation. Approaches for software validation divided in the fields dynamic analysis and static analysis are discussed. Furthermore, methods and techniques for both approaches are described.
- Chapter 4 describes the architecture of the Semantic Validation Framework. It includes the classification process and its results. For each metric category the semantic validation process is shown.

- Chapter 5 revisits the approaches introduced in Chapter 3. It is examined which of these approaches are suitable to validate metrics of the category non-deterministic. The result of this process is presented and the solution is explained in detail.
- Chapter 6 is the description of the prototype development and implementation. Correctness criteria for metrics are derived and validated using the solution found in Chapter 5. These criteria are integrated in the prototype.
- Chapter 7 consists of the evaluation of the prototype and a discussion about the overall results of the thesis. The prototype is tested with a new version of OPM. Necessary adaptations are explained.
- Chapter 8 concludes the document and possible future work is described.

2 OPM Environment and Problem Description

The goal of this thesis is the development of a Semantic Validation Framework for the IBM InfoSphere Optim Performance Manager. The framework is used to check the correctness of OPM. This means, it validates the metrics computed by OPM.

In this chapter the underlying project environment is introduced. It includes a description of the way OPM metrics are tested at the moment and reasons for the need of the Semantic Validation Framework. Known problems of semantic validation in case of OPM are discussed. In the last section the architecture of the framework is presented.

2.1 The Project Environment

The project environment consists of the IBM InfoSphere Optim Performance Manager itself, the test environment and an IBM internal tool for producing workload.

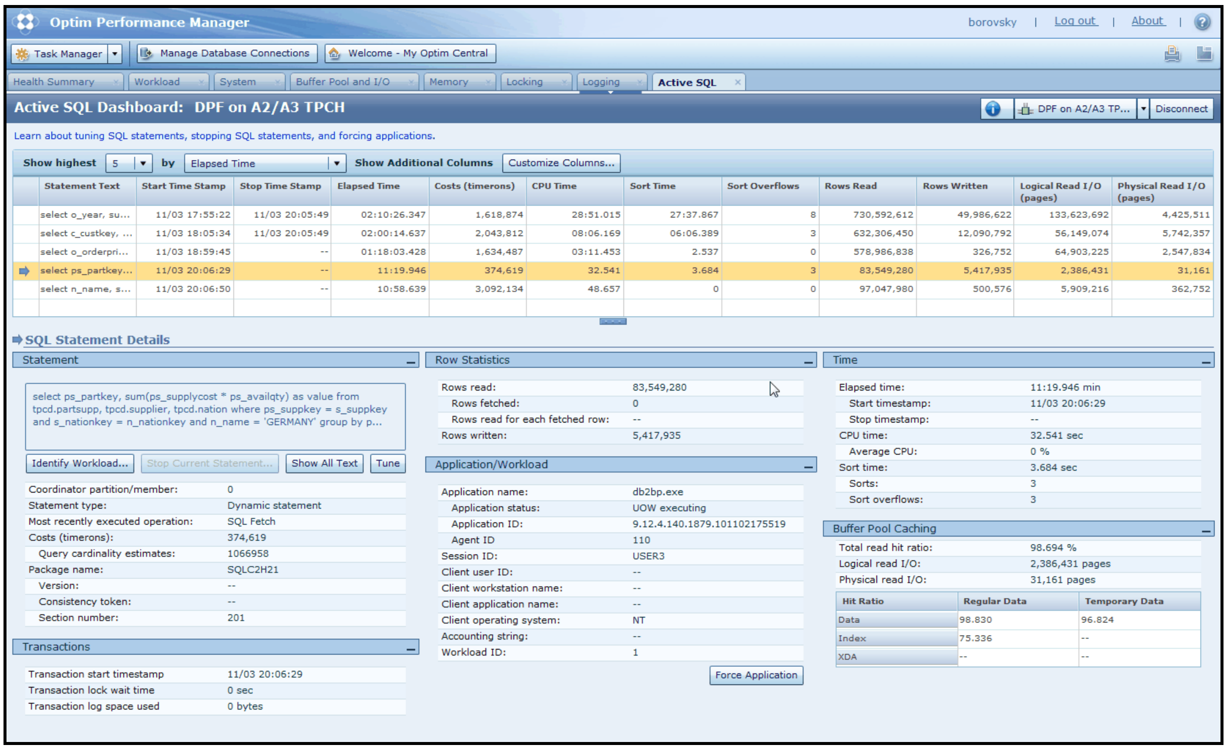
2.1.1 The IBM InfoSphere Optim Performance Manager

The IBM InfoSphere Optim Performance Manager is a performance analysis tool for DB2 databases on Linux[4], UNIX[5] and Windows[6] systems. OPM is able to identify, diagnose, solve, and prevent problems on monitored databases. On that account OPM collects, aggregates, and calculates performance metrics of DB2 databases and visualizes this data in a web interface.

The web interface serves as a tool for users to add and configure databases for monitoring, using predefined or custom monitoring profiles and also to administrate OPM in general. It contains different dashboards showing specific database contexts. There are dashboards including overall information about the health status of monitored databases. The so called inflight dashboards drill down to more detailed

[4] Trademarks of Linus Torvalds in the United States, other countries or both
[5] Trademarks of The Open Group in the United States and other countries
[6] Trademarks of Microsoft Corporation in the United States, other countries, or both

information about potential performance issues. In this dashboard category users can see data about locking, logging, workload, etc. In Figure 2.1.1 the active SQL Dashboard is shown. This dashboard includes information about the active SQL statements. It provides, among others, the statements itself, runtime, caused overflows, and how many rows are read. Users are able to set timeframes for which they want to see data. It is possible to analyze what events are happening at the moment and also what events have happened in the past.



Active SQL Dashboard: DPF on A2/A3 TPCH

Learn about tuning SQL statements, stopping SQL statements, and forcing applications.

Statement Text	Start Time Stamp	Stop Time Stamp	Elapsed Time	Costs (timerons)	CPU Time	Sort Time	Sort Overflows	Rows Read	Rows Written	Logical Read I/O (pages)	Physical Read I/O (pages)
select o_year, su...	11/03 17:55:22	11/03 20:05:49	02:10:26.347	1,618,874	28:51.015	27:37.867	8	730,592,612	49,986,622	133,623,692	4,425,511
select c_custkey, ...	11/03 18:05:34	11/03 20:05:49	02:00:14.637	2,043,812	08:06.169	06:06.389	3	632,306,450	12,090,792	56,149,074	5,742,357
select o_orderpri...	11/03 18:59:45	--	01:18:03.428	1,634,487	03:11.453	2.537	0	578,986,838	326,752	64,903,225	2,547,834
select ps_partkey...	11/03 20:06:29	--	11:19.946	374,619	32.541	3.684	3	83,549,280	5,417,935	2,386,431	31,161
select n_name, s...	11/03 20:06:50	--	10:58.639	3,092,134	48.657	0	0	97,047,980	500,576	5,909,216	362,752

SQL Statement Details

Statement

select ps_partkey, sum(ps_supplycost * ps_availqty) as value from tpcd.partsupp, tpcd.supplier, tpcd.nation where ps_suppkey = s_suppkey and s_nationkey = n_nationkey and n_name = 'GERMANY' group by p...

Coordinator partition/member:	0
Statement type:	Dynamic statement
Most recently executed operation:	SQL Fetch
Costs (timerons):	374,619
Query cardinality estimates:	1066958
Package name:	SQLC2H21
Version:	--
Consistency token:	--
Section number:	201

Transactions

Transaction start timestamp	11/03 20:06:29
Transaction lock wait time	0 sec
Transaction log space used	0 bytes

Row Statistics

Rows read:	83,549,280
Rows fetched:	0
Rows read for each fetched row:	--
Rows written:	5,417,935

Application/Workload

Application name:	db2bp.exe
Application status:	UOW executing
Application ID:	9.12.4.140.1879.101102175519
Agent ID	110
Session ID:	USER3
Client user ID:	--
Client workstation name:	--
Client application name:	--
Client operating system:	NT
Accounting string:	--
Workload ID:	1

Time

Elapsed time:	11:19.946 min
Start timestamp:	11/03 20:06:29
Stop timestamp:	--
CPU time:	32.541 sec
Average CPU:	0 %
Sort time:	3.684 sec
Sorts:	3
Sort overflows:	3

Buffer Pool Caching

Total read hit ratio:	98.694 %
Logical read I/O:	2,386,431 pages
Physical read I/O:	31,161 pages

Hit Ratio	Regular Data	Temporary Data
Data	98.830	96.824
Index	75.336	--
XDA	--	--

Figure 2.1.1: Active SQL Dashboard [2]

Furthermore, with the Extended Insight feature of OPM it is possible to monitor clients and applications (Java[7], WebSphere[8] Application Server, DB2 Call Level Interface, etc.), which execute workload on monitored databases. In this case, the collected data is called end-to-end data. With Extended Insight the user has the opportunity to analyze, for example, the runtime of SQL queries of different clients

[7] Trademarks of Oracle and/or its affiliates
[8] Trademarks of IBM in USA and/or other countries

and is able to find out which clients or applications influence the performance of the whole database system.

Beside the manual observation of the performance data by the user, OPM can alert users automatically, for example via email. The configuration of automatic notification is also done in the web interface. These automatic alerts reduce the response time of database administrators, thus issues can be detected and solved before they have big impact on the system. Additionally OPM provides reports for events which can be scheduled and are generated automatically.

From the technical point of view OPM consists of three major components (see Figure 2.1.2).

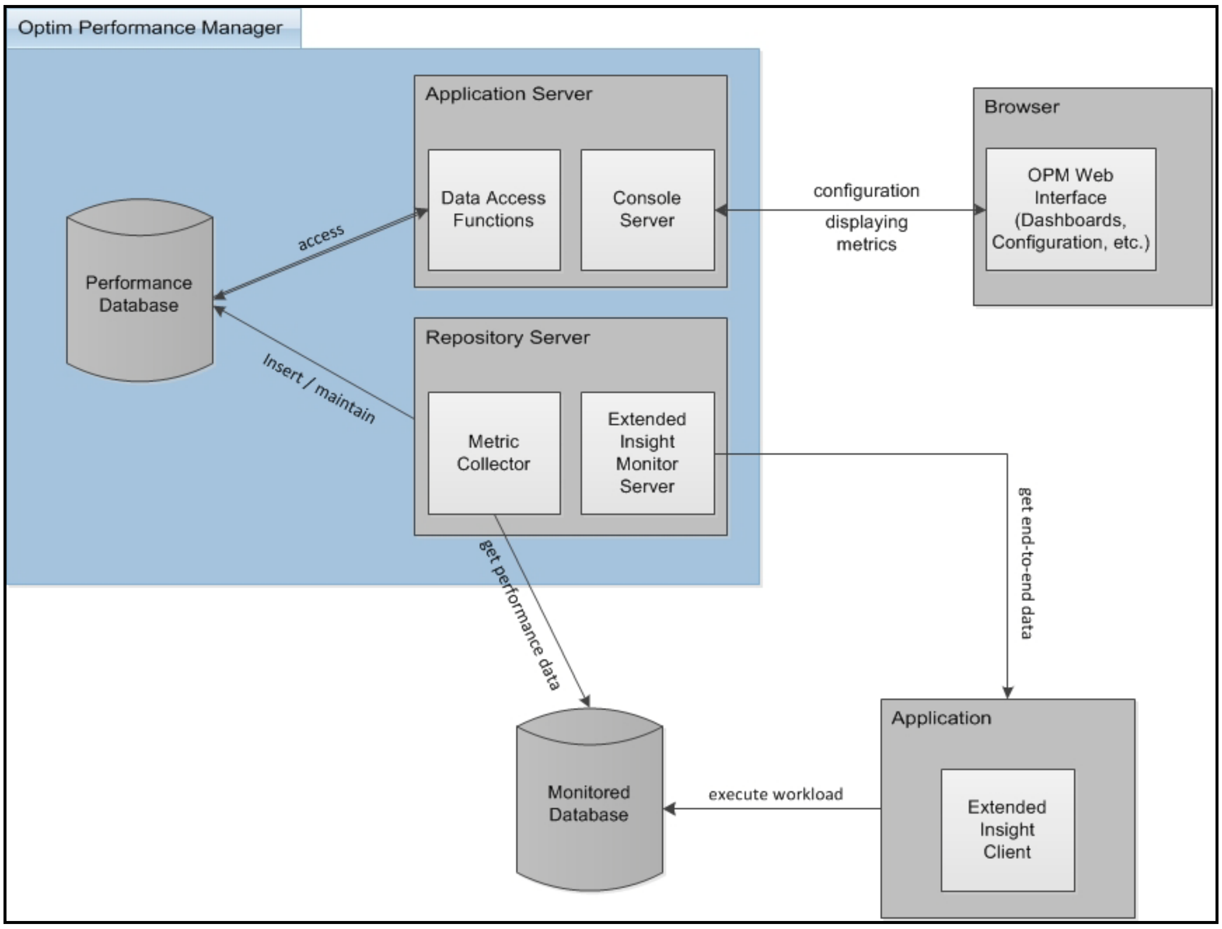


Figure 2.1.2: OPM Architecture (including Extended Insight)

- The Repository Server which collects the performance data from monitored databases and applications, using snapshots and DB2 event monitors.

- The Performance Database contains the performance data collected by the Repository Server.
- The Application Server which loads the performance data from the Performance Database into the web interface.

Between the Performance Database and the Application Server there are data access functions to access the data. These functions, used by the Console Server, serve as backend for the web interface of OPM and consist of two parts:

- The first part is used to set up and configure database monitoring.
- The second part is used to retrieve data from the Performance Database for the OPM dashboards. Additionally it computes further metrics based on the data from the Performance Database.

The procedures of OPM are as follows. After the configuration of monitoring, the Repository Server of OPM collects data from configured DB2 databases and/or database applications (for example SAP). Monitoring applications is only possible if Extended Insight is activated and the Extended Insight Client is installed on the application side. This data is stored in the OPM Performance Database and the Application Server accesses it using the functions mentioned above. Finally, the Application Server loads the data into the web interface where it is visible to the user. Now, the user is able to supervise the performance of his database systems and react when issues occur.

In the next section the available test environment is introduced. Further, the terms *run* and *iteration* in case of the test environment are defined.

2.1.2 The Test Environment

The test environment is an in-house developed tool. During test phases of OPM (for instance function verification tests or regression tests) it is utilized to test the data access functions and to check the integrity of metrics. For example, it verifies that metrics which constitute a value in percent are within the range 0 to 100. During the tests no user interaction is needed.

The test environment is also able to run an IBM internal workload tool (see Section 2.1.3) to automatically execute workloads on databases. Figure 2.1.3 pictures the OPM architecture including the test environment. The test environment replaces the Application Server and the web interface.

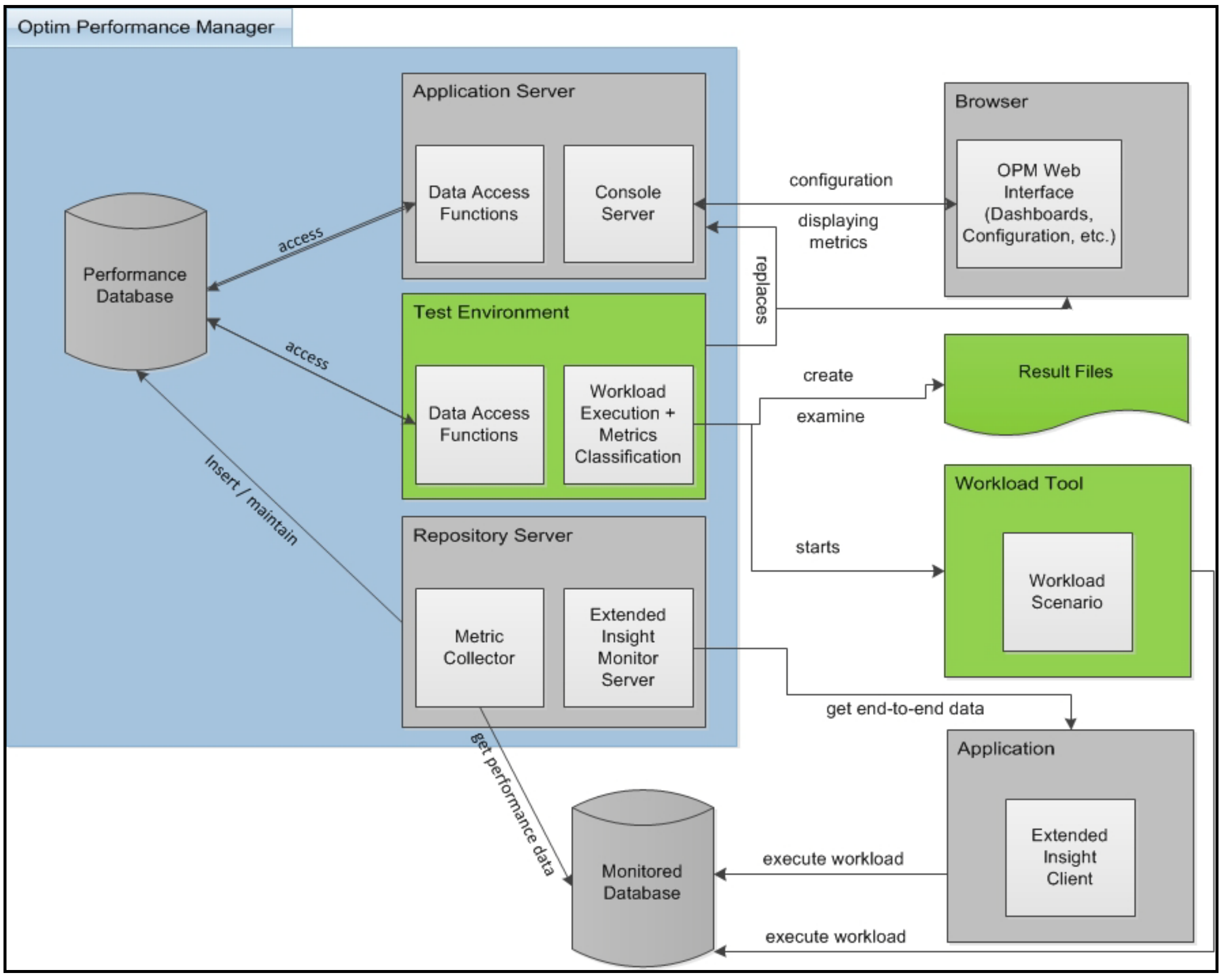


Figure 2.1.3: Test Environment to classify the performance metrics

It makes use of the data access functions in the same way as the Application Server of OPM. This includes adding databases to OPM, activating the monitoring, loading the performance data from the database, and computing it further. After the process of loading and computing, the data is the same as represented to the user in the web interface and is stored in result files. With these tests various scenarios in the area of configuration and data retrieval can be validated.

A *run* of the test environment consists of a number *n* of *iterations* defined by the user and ends with the automatic investigation of the result files. Figure 2.1.4 shows a run and the single steps in each iteration.

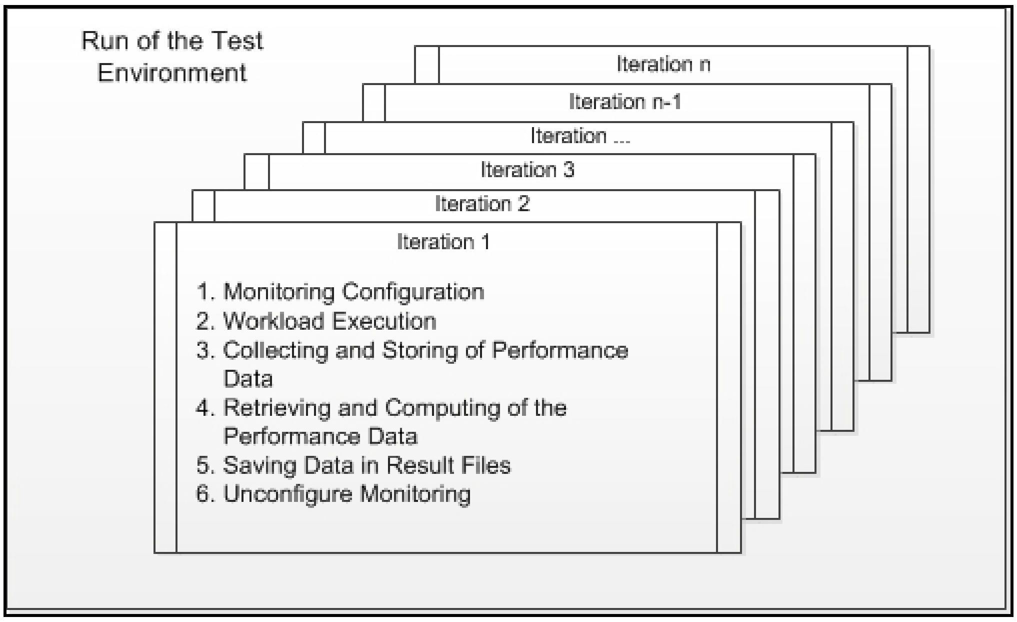

Run of the Test Environment
Iteration n
Iteration n-1
Iteration ...
Iteration 3
Iteration 2
Iteration 1
1. Monitoring Configuration
2. Workload Execution
3. Collecting and Storing of Performance Data
4. Retrieving and Computing of the Performance Data
5. Saving Data in Result Files
6. Unconfigure Monitoring

Figure 2.1.4: One Dedicated Run of the Test Environment

In each iteration a database for monitoring is added and configured. Then workload is generated on the monitored database using the workload tool. After the workload execution is finished and OPM has collected the performance data, the test environment stores the data in the result files for additional investigations.

These results of a run of the test environment are Excel files containing the values for every metric, the formula how the metric is computed, and various other information about the metric.

The test environment offers a simple first step to semantic validation. A function is available, which examines the result of a run and is able to classify every metric in one of the three categories deterministic, semi-deterministic or non-deterministic. This process is called classification. The idea is that further semantic validation differs depending on the behavior (the category) of a metric. This is explained in more detail in Section 2.2.2. The classification provided by the test environment is based on the following decision process:

- For every metric it is checked if all values of all iterations are the same. If this is the case, the metric is sorted into the category *deterministic*.
- If a metric has different values for the iterations and these values are numbers only, the average value, the maximum value and the minimum value can be computed. If the maximum and the minimum value do not exceed a predefined deviation of the average value, the metric is categorized *semi-deterministic*.
- All metrics which do not fit in the pattern above are classified *non-deterministic*. This means these metrics attain different values in the iterations and they exceed the range predefined by the user. Another possibility is that these metrics allocate values which are no numbers and differ in the iterations. For this kind of values no average value can be computed and so they are marked non-deterministic.

At the end of this analysis, every metric is categorized exactly to one category. Though, this functionality is already available only few tests concerning the classification have been done before the thesis started. This means that the classification is an important step in this thesis for the development of the Semantic Validation Framework.

As mentioned above, the test environment is able to run a workload tool to execute workloads on the monitored databases. To do a classification this workload has to meet several requirements. These requirements and the procedure of running workload are described in the next section.

2.1.3 The Workload Tool and Workload Requirements

The workload tool is used during the test phases of OPM to simulate different scenarios on monitored databases. Among others, this includes common data manipulation and retrieving but also deadlocks or sorts can be triggered to cause all kinds of events on the database. These events are monitored by OPM.

The workload tool uses so called workload scenarios. These scenarios are encoded in Extensible Markup Language (XML) and contain SQL statements which are executed on the monitored database. Workload scenarios consist of three phases:

1. **Preparation**: Bufferpools, tablespaces and tables are created. Tables are filled with data.
2. **Execution**: SQL statements are processed which perform different types of data manipulation on the created objects and data retrieval.
3. **Cleanup**: All created objects are deleted.

The used workload has to meet several requirements to be able to classify the performance metrics and serve as workload to derive correctness criteria as well.

- It should run for a finite time period because the test environment is waiting until the workload tool has finished its execution before it proceeds. This guarantees comparable results for a specific workload since the performance data retrieval starts every time as soon as the execution of the workload is finished. The workload tool is also used outside of the test environment. In this case the workload is running in an infinite loop as long as the user does not stop it manually. This is not possible when using the test environment.
- It should be easy to adjust a workload because it may be possible that metrics have a different behavior if the workload is increased or decreased. Increasing or decreasing a workload means that the number of statements and/or the data volume is changed.
- It should cover as many metrics as possible. This means that the SQL statements executed by the workload create data, which should affect preferably all metrics.

The first and the second requirement can be achieved without big effort. Using the workload scenarios has the advantage to benefit from structures provided by the XML schema of the workload tool, for example if-statements or loops. This makes it possible to adjust the workload easily and limit the runtime. Furthermore variables can be used, for example for tables and inserted values.

The third requirement is much harder to be accomplished and increases the probability to end up in more complex scenarios. This could mean that for these scenarios adjustments are more difficult to be implemented and the effect of adjustments are harder to recognize. To avoid this complexity several scenarios with different coverage are used in this thesis. These scenarios are developed during the classification process.

One example of these scenarios is “*Massive Objects*”. It creates user defined bufferpools, tablespaces, and tables, and runs SQL statements, which cause activity on the created objects. This activity consists of SQL statements inserting data into the tables and updating this data. Further, the scenario executes statements, which select or delete data, for example. To make it easier to adjust the workload, the scenario has input parameters. The Figure 2.1.5 shows the input parameter for the number of tables which should be created during preparation phase.

```
<arg default="10" description="The number of tables to create."
key="tbl" type="int"/>
```

Figure 2.1.5: Example for Input Parameters of Workload Scenarios

In this example, the workload can be modified by changing the value for the attribute “tbl”. This means, changing the number of tables, which are created and used during the execution of this scenario. Another possibility to change the workload is to increase the number of loops for the insert-, update-, select- and delete-statements. As a result more database accesses are performed.

In the Section 2.2 the motivation and reasons for the need of the Semantic Validation Framework are described. Further, known problems and possible difficulties are pointed out.

2.2 Problem Description

The Semantic Validation Framework is meant to be an extension of the test environment. Its goal is to enlarge the testing capability of the environment and provide assertions about the correctness of metrics to the user. These assertions are created automatically without user interaction. As an extension of the test environment the Semantic Validation Framework is able to make use of its functions. This means, configuring the monitoring of OPM, running workload, data retrieval and saving of the data in result files is already available.

This section explains why it is desirable to have such a framework for validating the OPM metrics. It shows the shortcomings of the actual testing approach and hints at the problems which may occur during the development.

2.2.1 Motivation and Reasons for the Semantic Validation Framework

The scope of the tests provided by the test environment is limited. It consists of finding exceptions during the invocation of the data access functions which retrieve the data and to provide simple means of data verification. For example, checking a percentage value as mentioned above.

Currently the question if the retrieved values are correct cannot be answered in a simple way. Most of the data verification tests have to be done manually. It is very critical to proof data correctness during testing, as errors found on customer side can damage the trustworthiness of OPM to a high degree. The question for data correctness may sound simple but is very complex at the second glance.

At the moment OPM is providing approximately 1000 different metrics. Manual tests in this case are very time consuming, expensive and error-prone because every metric has to be checked individually. Further, to be able to draw assertions about the correctness, it is necessary to understand the metric itself. This requires the skill set of an experienced database administrator.

To prove the correctness of a metric, which is not computed further, the value in the Performance Database has to be compared with the actual result. For metrics which are computed of several values, all these values have to be verified in the

Performance Database, the computation has to be done manually and the results have to be compared. In practice this is impossible for this high number of metrics.

Automating these comparisons of the values in the Performance Database with the actual results is not a solution either. During development the way metrics are computed changes. This includes changes in the schema of the Performance Database. For example, names of tables, attributes are adjusted or data is summarized in new tables. To prove correctness, all these changes have to be considered and implemented in the automatic tests. For every new version it has to be checked if adjustments of the tests have to be done. In development phase a new build for testing is published nearly every day, making this approach impractical.

The goal of the Semantic Validation Framework is to provide a tool for automatic correctness proving, covering all metrics. No more user interaction after starting the test should be required. It has to avoid the problems mentioned above. This means changes in the Performance Database or in the data access functions should not influence the framework. The ongoing development of new versions of OPM makes it necessary for the framework to be adjustable in a simple way. For example adding new metrics or removing metrics which are not needed anymore. It should be possible to run these tests every day by different persons. This requires that the tests are fast, uncomplicated and the results are meaningful.

The next section discusses the problems occurring for semantic validation in case of OPM and how the different behavior of the metrics influences the process of finding correctness criteria.

2.2.2 Problems of Semantic Validation

The basic problem of semantic validation (see Definition 3.1.2 in Section 3.1) is to find criteria which allow to draw conclusions about the correctness of the tested software. These criteria depend on the nature of the objects, which have to be validated (in this case the OPM performance metrics).

There are different patterns of behavior concerning the assumed values of the metrics. The range which these values can attain depends on the workload on the monitored database and the nature of the metric. The values are influenced by the system of the monitored database, for example, by the load factor or network speed. This means that for some metrics the monitored values differ when monitoring the same workload more than once. As a consequence, the criteria for the correctness of these metrics are different from the criteria for metrics which values do not deviate for the same monitored workload.

There are three categories of OPM metrics (described in Section 2.1.2). Each category describes a certain behavior of the values a metric can attain (output) when the same workload is monitored under the same circumstances several times (input). The input consists of the workload and the parameters of the underlying system of the monitored database. These parameters include among others DB2 parameters, DB2 version and preferences of the server.

This means that the category of the metrics affects the difficulty finding appropriate relations between the input and output values. It is considered that developing correctness criteria for deterministic and semi-deterministic metrics is easier than for non-deterministic metrics. Deterministic metrics follow explicit input-output rules which can be derived with little effort.

Finding correctness criteria for semi-deterministic metrics is more complicated. The output values of metrics of this category can vary over a finite range for the same input. First problems occur when trying to define a range within all values should be located. For example consider a program running on a normal desktop computer, computing the duration of SQL queries. The found range in which the computation is still correct is for example 5% variance of the average value. If the 5% range is exceeded it would mean that a defect occurred in the function calculating the duration. The duration of queries depends among others on the hardware and processes, which are running on the computer while the queries are executed. So if running the same queries on a much more powerful system the 5% range could be too large and defects would not be found. So the metric is still semi-deterministic but the range has to be adjusted according to the environment. This could mean specifically for OPM that running the Semantic Validation Framework on different test machines leads to adjusting the ranges of correctness for each machine or accept fuzzy ranges.

For non-deterministic metrics the output for the same input can vary within an infinite range. Proving correctness for this kind of metrics needs different testing approaches compared to the other categories.

In the next chapter, semantic validation and semantic correctness are defined. An overview about existing techniques and methods for software verification, which can be suitable for the framework, is provided. It includes two different strategies and the respective approaches.

3 State of the Art

Approaches for software verification can be divided into two fields, *dynamic analysis* and *static analysis*. The difference between these verification techniques is the way how they deal with the software under test itself. In dynamic analysis the software is executed and the output is analyzed. Static analysis does not execute the software but analyzes the structure of the software by checking if certain conditions are fulfilled. Contrary to dynamic analysis, which checks individual runs only, static analysis is able to prove the absence of errors because it examines all states and paths of the software [3].

In order to be able to ponder if an approach is suitable for semantic validation in the case of the IBM InfoSphere Optim Performance Manager, first a definition for semantic validation for this thesis is given. Then the different techniques are introduced.

3.1 Definition of Semantic Validation

In general, semantics is the study of meaning [4]. More specifically, semantics deals with the relation between symbols in languages and their denotation in the nonlinguistic world. In contrary, syntax describes how expressions are formed with the symbols of the language [5]. In computer science semantics on the one hand is understood as a mathematical model for programming languages, which helps to understand the performance of programs. This semantics of programming languages or *formal semantics* consists of three types [6].

- First, the *operational semantics* which describes the meaning of a programming language by transition-functions from one state to another state.
- Second, the *denotational semantics* which is rather mathematical using partial orders, continuous functions and least fixed points.
- Third, the *axiomatic semantics* which is making use of different assertions which have to be satisfied before and after the execution of a program or a function.

On the other hand, semantics deals with the meaning of symbols. If a human being has a description of a program in natural language telling what the program is supposed to do, he is able to understand the semantics of this program. For example if the description of a single function is “number of users” everyone understands that this function is counting the users.

The mathematical models of formal semantics provide tools to reason if a program is implemented semantically correct. Semantically correct means that the results of the program are appropriate in their meaning, the general understanding of semantics. Referring to the example above the result for “number of users” can never be semantically correct if it is a negative value. Consequently semantic testing in terms of software is a methodology to verify the relationship between the data produced by this software and its semantic correctness.

The data produced by the software depends on the data which has been used as input for the software. This means that the semantic correctness of software is also dependent on the input which is used. In more detail, the input can be considered as preconditions and the output as postconditions. Before a run of the software the preconditions have to be valid and after the run the postconditions have to be valid. This leads to the following definition of semantic correctness for this thesis.

Definition 3.1.1 Semantic Correctness of Software

Software is semantically correct if the postconditions according to the preconditions are fulfilled:

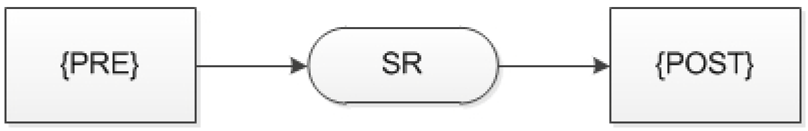


Figure 3.1.1: Semantic Correctness of Software

Where {PRE} is the set of preconditions, {POST} is the set of postconditions and SR symbolizes a run of the software. So if {PRE}SR{POST} is valid, meaning that {PRE} is fulfilled before the run and {POST} is fulfilled after the run, the software is semantically correct.

The definition of semantic correctness of software automatically leads to the definition of semantic validation.

Definition 3.1.2 Semantic Validation in Terms of Software

Assuming the definition of semantic correctness, semantic validation in terms of software is the process of verifying the set of postconditions according to the set of preconditions of the software.

In Sections 3.2 and 3.3, methods will be introduced which are capable for semantic validation in terms of software. The first part will cover methods based on mathematics and formal semantics belonging to static analysis and the second part will contain empirical methods belonging to dynamic analysis.

3.2 Static Analysis - Formal Methods

Formal methods are a practice to specify a system and its desired properties, for example functional or temporal behavior, by languages and techniques which are based on mathematics. It is possible to express what a system should do in a mathematical way and then check automatically if the system is compliant within this specification. Formal methods serve to verify the relationship between the source code and the meaning of the implemented function. For example one approach to test this issue are assertions between input variables and system variables. These variables correlate before, during and after the execution of a program or single function and so have to fulfill certain constraints [7][8]. In the next section, model checking and theorem proving will be introduced, two approaches using formal methods to verify software systems.

3.2.2 Model Checking

Model Checking is a technique to automatically achieve a formal verification that a system runs according to its specification. In Figure 3.2.1 an overview of the process of model checking is shown. The idea behind model checking is to map the original system under test to a model, which is able to serve as input for a model checker. This model has the same characteristics and behavior as the original system but it is represented with methods of mathematics or formal semantics. For example, systems can be modeled by finite state machines or transition systems. In these formal representations the nodes stand for the system states and the transitions between the nodes symbolize the possible state changes.

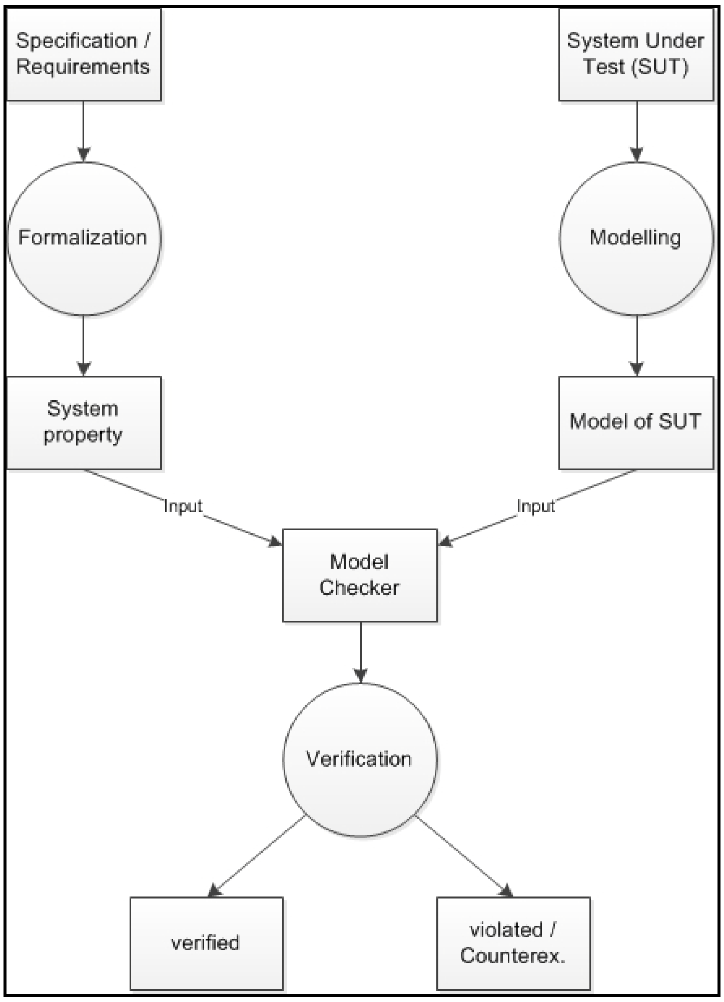


Figure 3.2.1: Model checking process

To be able to check the model against the specification automatically, the specification has to be existent in a formal description as well. One common set of logics to formalize the specification of a system is the set of temporal logics. Temporal logic allows expressing properties in different instances of time using temporal operators [9]. For example, it is possible to specify that a property has always to be fulfilled, meaning in every state of the system, or a property has to be valid only in the next state. Especially linear temporal logic (LTL) and computation tree logic (CTL) or modified versions of LTL and CTL are common in the field of model checking.

One big advantage of temporal logic is that formulas can be translated into automata. If the specification and the model are available as automata the process of checking that the model fulfills the specification is simplified to the comparison of two automata. This leads to efficient algorithms in model checking based on emptiness checks [10]. In short, the process of proving correctness starts with building the automata of the model and the negation of the specification which should be checked. The next step is to build the product of these two automata and test if the resulting automaton accepts any word. If it accepts a word, the actual property is not fulfilled. If it accepts no word, meaning that the language accepted by the automaton is empty, the property is valid in the system.

After the specification and the system are available in some kind of formal description, they serve as input to model checkers. Model checkers prove for a given system and specification, whether the system runs accordingly to its specification or if there are violations. If violations are detected, a counterexample is provided to help finding the error in the real system.

The common procedure of model checkers to verify systems is to check for every reachable state of the system model if the requirements are met. Reachable states are all states which are accessible through a sequence of state changes starting in the initial state. To make sure that the model checker does not test a state more than once, all states have to be kept in memory. This leads to the so called *state space explosion problem* and causes immense costs of memory [11].

However model checking does have a major advantage. Model checking verifies a large set of possible paths (test cases) in one run. In order to benefit of this advantage many approaches to solve the state-space explosion problem have been presented in the last years.

- For example it is possible to divide the state space and test the requirements against every state partition. So only one partition has to be kept in memory [12].
- Another method to deal with the state space explosion is to store for every successor state the changes compared to its predecessor. This method promises a memory reduction of about 95% [11].
- A different procedure is implemented by the open-source model checker SPIN using on-the-fly verification [13]. Consequently SPIN mostly does not need to construct the whole state space because it constructs the next state only if needed. SPIN is available since 1991 and based on the automatic theoretic approach by Moshe Y. Vardi and Pierre Wolper saying that for every temporal formula (SPIN uses LTL) it is possible to construct a corresponding automaton which accepts exactly the language specified by this temporal formula [14]. This results in emptiness checks of two automata as described above.

Theorem proving is another approach of static analysis. It does not need a model but it checks certain properties of the program, which have to be valid. This approach is introduced in the next section.

3.2.3 Theorem Proving

Theorem proving is a mathematical methodology to prove if a function matches its specification. The basic idea for theorem proving is to check that a specific property is satisfied before the execution of the program (precondition) and another specific property is satisfied after the run of the program is finished (postcondition). If the check is successful the correctness of the program is proved. Generally the properties can be seen as “if-then” statements. “If” certain conditions, for example input values or system states, are fulfilled, “then” after the computation the system will end in a corresponding state and has a corresponding output [15].

Theorem provers, which check these kinds of assertions, are most of the time based on the calculus by Hoare providing axioms and inference rules to reason that a precondition implies a postcondition. Using axioms and inference rules to decide that a program is implemented correctly, leads to the big advantage compared to model checking: Theorem proving does not need to verify every possible state of a system, hence it is capable of dealing with an infinite state space [16].

A slightly different idea but still based on logic is to use the rules provided by a calculus several times on the formula which has to be proved, until a valid formula is derived or no rules can be deployed anymore. If no more rules are valid to use, the formula is confuted [17].

Another way to implement theorem proving is to model the system as a finite state machine and the theorem as input values and an actual state, leading to an output value or a sequence of output values and a successor state. To prove the theorem it has to be checked if the finite state machine gives the desired output and stops in the desired state [15]. The difference to model checking in this case is that theorem provers again do not check every possible state.

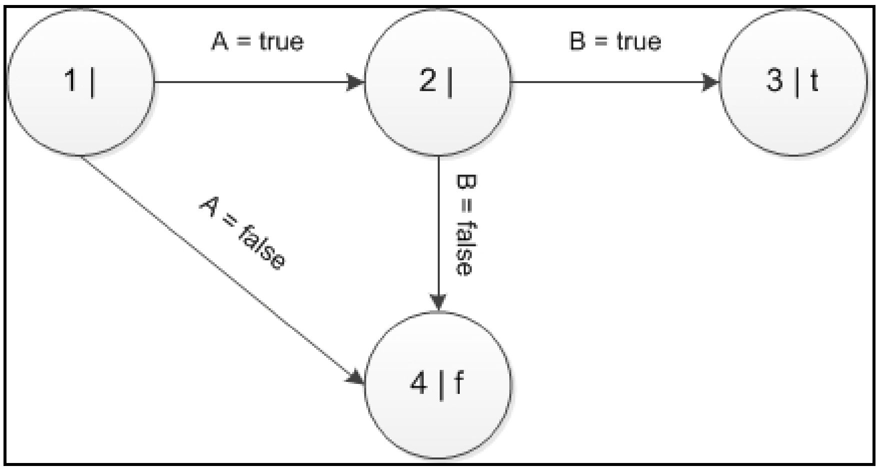


Figure 3.2.2: Example of a Finite State Machine

An example for this way to prove a theorem can be seen in Figure 3.2.2. The finite state machine ends in state 3 and outputs “t” if A and B are both true else it ends in state 4 and outputs “f”. A suitable theorem is the following: ((S(1), {true, true}) → (S(3), t)) where the set {true, true} is the input and “t” is the output. A theorem prover now checks what happens when the input is {true, true} and the system is in state 1 and compares these results with the right side of the theorem.

3.3 Dynamic Analysis

Unlike static analysis, which has been introduced in the previous section in terms of formal methods, dynamic analysis is rather based on empirical methods and statistics than on mathematics or formal semantics. Dynamic analysis tries to verify software by executing instead of looking at the structure only. For example there are methods which check certain conditions during runtime or techniques comparing the results of runs with the input or with results of previous runs. The following sections describe some of the existing dynamic analysis methods.

3.3.1 Testing

In the field of software, testing is the execution of the software under certain circumstances to prove correctness and completeness. These circumstances are called test cases and often refer to the input data. Depending on what kind of software or which components are tested, test cases can be sequences of user interactions or well-defined steps to be executed, for example the testing of user interfaces. In the following section, test cases always have the meaning of input values. Test cases allow to draw conclusions whether the software is implemented correctly according to its specification [18]. To do this an *oracle* has to exist.

An oracle can be understood as a method to decide whether a test case succeeded or not, by comparing the connection between input and output. Today in industrial practice most of the time these oracles are humans, checking the output manually. Because of the possible very wide range of output values, generating oracles for automatic verification is the big challenge beside the choice of test cases in software testing [19].

In the following, existing testing techniques are introduced. The single test methodologies can be distinguished by how they deal with the actual code of the software.

- **Black box testing** is a method which does not consider the code but only the input and output values.
- **White box testing** takes a closer look at the code itself, checking also paths and states of the system. Therefore, detailed knowledge about the implementation of the system under test is required.
- **Grey Box testing** combines these two methods and makes it possible to analyze values during computation [18].

In addition, software testing techniques differ in how they choose test cases. Unlike methods of static analysis, testing in general is neither able to check every state of software systems nor every possible path. In testing, the chosen test cases affect which states and paths are checked. The strategy to derive test cases has a big impact on what defects can be found and in which area of the tested systems the defects will be detected. Two possible techniques to choose test cases are introduced here:

- **Random Testing:** One strategy to select test cases is choosing the test cases randomly from the input domain. This strategy can be very helpful if there is no information about how to contain the input. Furthermore, since no human interaction is needed for building test cases, nobody intended or accidently can falsify the input. In addition, random testing due to the wide range of output values, needs an automatic oracle to test if the system behaves properly [19] [20]. To make sure that the testing covers as much code as possible random testing has to be adjusted, leading to more predetermined strategies of choosing test cases.

- **Sub-domain Testing:** In sub-domain testing, the input domain is divided in sets of values which are considered to be related in some way. For example one way to identify sub-domains is to deduce them directly from the software specification. Sub-domain testing belongs to the *systematic testing* techniques but it still has random elements. In fact the test cases are chosen from each sub-domain to achieve wide coverage but the selection in a sub-domain is done randomly. There are some modifications for sub-domain testing, e.g., trying to maximize the distance between the input values or keeping the distance at the same level with the goal to cover a preferably wide range. Other approaches propose to choose the values in a way they will be used in reality [21].

Beside the way test cases are chosen, testing methods can be distinguished by foreknowledge about the output values. The methodology is depending on the knowledge of the output behavior.

- **Statistical Testing:** Statistical testing is used if the output of the software under test is stochastic. This means that for the same input different output values are possible. In this case, statistical testing is a possibility to determine that the behavior of software is correct, at least with a certain probability. Testing still refers to comparing input values to output values, but in this case deriving an exact oracle is impossible because of the stochastic distribution of the output. Having test cases with known correct output a probabilistic distribution can be calculated from these values. Otherwise, reference values can be derived from running test cases and proving manually if the results are valid. Based on these reference values the expected distribution can be established [22]. This allows to compute the probability that the output of test cases is correct.

- **Deterministic Testing:** In contrast to statistical testing the expected output in deterministic testing is known. This means for every input or test case the corresponding output can be determined according to the specification [23]. As a consequence it is possible to make exact statements about the correctness for the test input and deriving an oracle can be done with less effort compared to the case of random output values.

Beside the method of software verification with the help of test cases, there are various other approaches in the field of dynamic analysis. For example, runtime assertion checking.

3.3.2 Runtime Assertion Checking

Runtime assertion checking is another technique which is verifying software during execution. Assertions are requirements for individual system states and are checked automatically when the system is running. Possible assertions are pre- and postconditions or invariants, for example. Preconditions have to be fulfilled before the execution of the program or single function, and postconditions after the execution. Invariants are requirements which have to be valid throughout the whole run. Assertions can be built from Boolean expressions and constraints and because they are checked during runtime, the location of an error is known immediately [24].

Verifying software by checking assertions is also done by theorem proving. Theorem proving is using inference rules or calculus to prove the correctness of the system under test mathematically, instead of executing the program. So in fact theorem proving covers all possible states of the system but assertions which are checked during runtime can be directly implemented in the code. This makes runtime assertion checking easier to understand and to develop than theorem proving.

3.3.3 Conclusions

In addition to the described approaches, there are ideas to merge static and dynamic analysis to combine the advantages of both approaches and reduce limitations. In static analysis the whole state space gets explored, including possible paths which never occur during runtime. Consequentially, defects are detected which do not affect the software quality and are time consuming to fix. In dynamic analysis coverage is a big limitation. Having an infinite state space it is impossible to check every path and even if the state space is finite, huge amount of test cases have to be provided to achieve a sufficient coverage.

One possibility to combine both, static and dynamic analysis is that static analysis discovers potential problems which are then verified using dynamic analysis methods [25].

Further, there are approaches which compute semantic relations between graph-like structures. They can be used to compare different inputs (for example database schemas) according to their semantics. One of these approaches is S-Match which calculates semantic relations between nodes of two trees which serve as input [26]. The results are the strongest relations between any pair of nodes of the input tress.

The next chapter describes the architecture of the Semantic Validation Framework. Furthermore, it contains the classification of the metrics and the enhancements of the test environment, which have to be done before the classification can be performed. The last part is the process of semantic validation for each metric category.

4 Semantic Validation Framework

Chapter 4 is divided in three parts. Part one explains the architecture of the Semantic Validation Framework. In part two the classification process and the results of the classification are described. Before the classification can be accomplished the test environment is enhanced to provide all necessary functionality. The available basic function for classification described in Section 2.1.2 is not sufficient. Part three provides the process of semantic validation in the framework for each category

4.1 Semantic Validation Framework Architecture

The Semantic Validation Framework is considered to draw assertions about the correct implementation of a metric. These assertions are presented to the tester. Figure 4.1.1 shows the architecture of the framework. The components marked in green are developed during this thesis. The OPM Environment in the figure consists of OPM, the database management system DB2 and the monitored database. In the figure it is assumed that the classification of the metrics has been accomplished. Every metric which is validated has been categorized in exactly one category.

The whole validation process of the OPM performance metrics is as follows: First, workload is executed on the monitored database. The created data is monitored by OPM. As soon as the execution of the workload is finished the test environment retrieves the data and stores it in result files. This procedure can be done several times to receive more data.

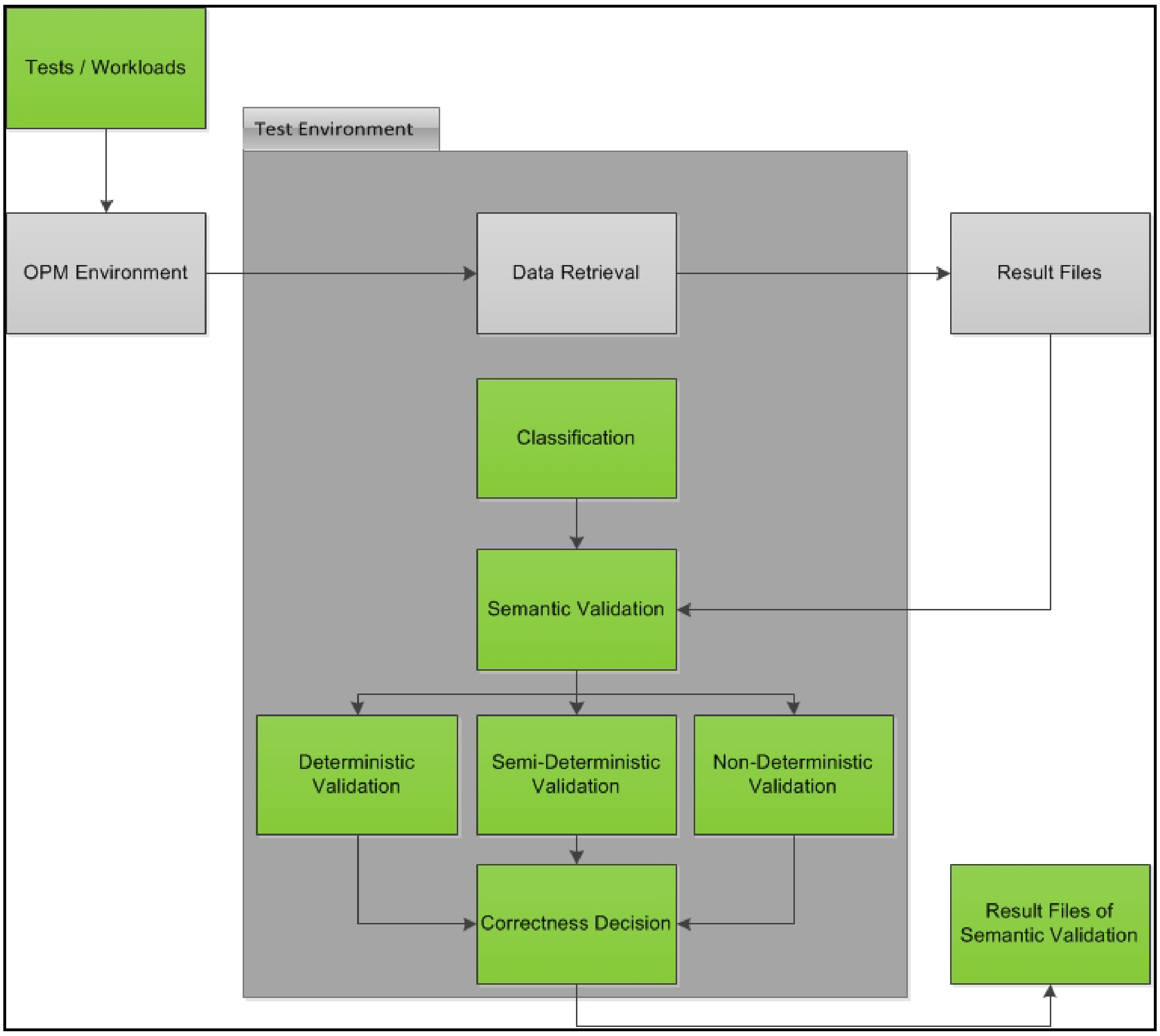


Figure 4.1.1: Semantic Validation Framework Architecture

After all tests are executed and the data is saved, the semantic validation starts. The first part of the validation is not affected by the metric category. For example the checking whether the actual category fits to the reference category is the same for every metric. The next part of the validation is based on the results of the classification. There is an individual validation process for every category. The results of these validation processes are decisions about the correctness for each metric. These decisions are presented to the user in a new result file.

The details of the classification process, the used workload and which changes, and improvements of the test environment have to be done, are explained in the following section.

4.2 Classification Process and Enhancements of the Test Environment

OPM collects a large number of performance metrics. In this thesis it is not possible to handle all of these metrics. Hence, the classification is done exemplarily for the workload dashboard. This dashboard contains 40 different metrics, including metrics about the system of the monitored database and metrics about various health indicators as failing transactions or sort performance. This means that all future validation in this thesis is done with metrics of the workload dashboard.

The classification of the metrics is the basis for the development of the Semantic Validation Framework. First, the test environment has to be extended and adjusted because it does not provide all needed functionality. The workload scenarios which are used to create the workload on monitored databases for every run of the test environment have to be developed and verified.

The workload should affect as many metrics as possible. If a metric is not affected, incorrect results will be retrieved for this metric. For example a metric which is not influenced by the executed workload has “0.0” values only. Theoretically this metric is categorized as deterministic. In the case of validation it will always be marked as correct when comparing reference values with actual values. If using a workload, which creates data for this metric, it may be classified as non-deterministic or semi-deterministic and a proper validation is possible.

In order to simplify the validation it is preferable to have as many metrics as possible in the categories deterministic and semi-deterministic. For these categories the validation process is simpler and more precise than for non-deterministic metrics.

To achieve this, a deeper understanding of every metric is required. This is needed to build the proper workload and to adjust the test environment. Experiments with different configurations of workload led to first insights and changes in the test environment. These are explained in the next section.

4.2.1 Changes and Improvements of the Test Environment

During first test runs it became clear that the existing version of the test environment does not contain all logics needed for the classification. The following adjustments had to be implemented:

1. Improved handling of “TIME_SERIES”
2. Implementation of parameter “lastMinutes”
3. Normalization of “TIME_SERIES”
4. Information added to the result files

There are two different aggregation types for metrics in OPM. For a selected timeframe metrics of the type “SINGLE_METRIC” have only one value. This timeframe can be chosen by the user and has to be at least one minute. Metrics of the type “TIME_SERIES” have a value for each one minute time interval because OPM collects data every minute. For example, if the timeframe is ten minutes, these metrics have ten values.

Metrics of the type “TIME_SERIES” have more than one value for each iteration of the test environment, depending of the duration of the workload. If the workload is running for 15 minutes, the metric has 15 values. For these metrics the classification process has been very simple so far. Up to this point a “TIME_SERIES” metric has been classified as deterministic if all values for every time interval and each iteration have been the same. Otherwise it has been categorized as non-deterministic.

For improving the classification of “TIME_SERIES” metrics they are now handled similar to the type “SINGLE_METRICS”. If the metric is not deterministic and consists of numeric values, the average value and the maximum deviation are computed for every time interval. The metric is classified semi-deterministic if all deviations are within the predefined range. In every other case it is classified non-deterministic.

Changing the intensity of workload it turned out that some metrics need a kind of “warm-up” time to reach a stable or nearly stable state. In this case, a stable state means that the value of the metric does not change anymore over the time if the workload executes the same statements in each time interval. Several factors may lead to such a behavior. One possible explanation is that it takes a certain amount of time until the bufferpools of DB2 are filled with the proper data requested by the workload scenarios. Further, the preparation phase of the workload is monitored. This means that the data retrieved in the beginning differs from the data during the actual workload execution.

One example of such a metric is shown in Figure 4.2.1, which is the number of all statements processed per minute. It can be seen that the values of this metric approach around 6000 when increasing the runtime of workload. The values on the x-axes are the number of loops of the workload scenario and in this case are corresponding to the runtime. In every loop different SQL statements e.g. SELECT- or INSERT statements are processed. Increasing the number of loops means higher workload. Without changing other parameters it ends up in a longer runtime because the number of loops processed each minute remains the same.

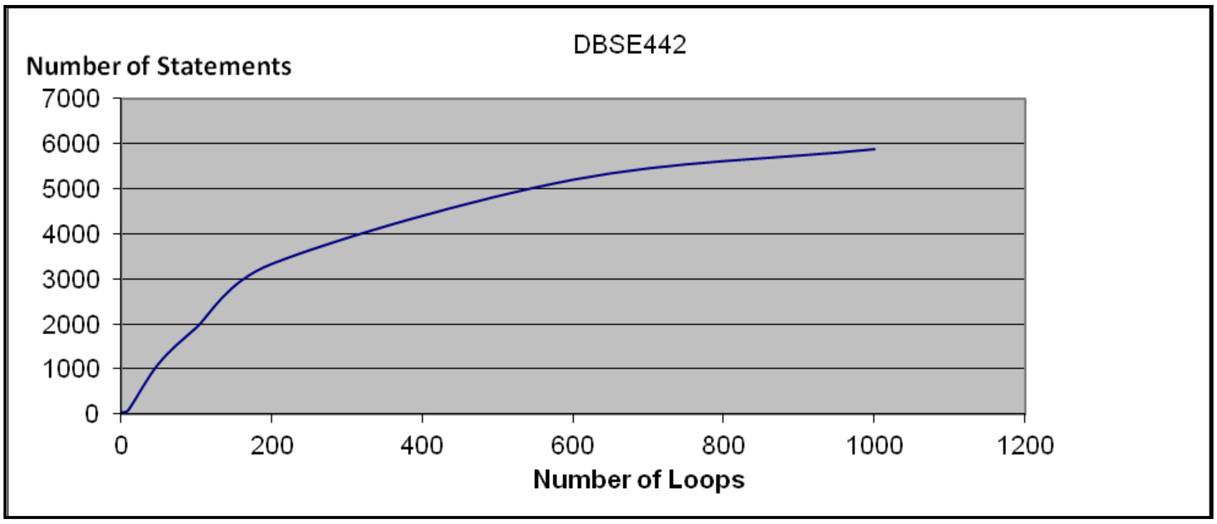


Figure 4.2.1: Warm-up Time for Metric DBSE422

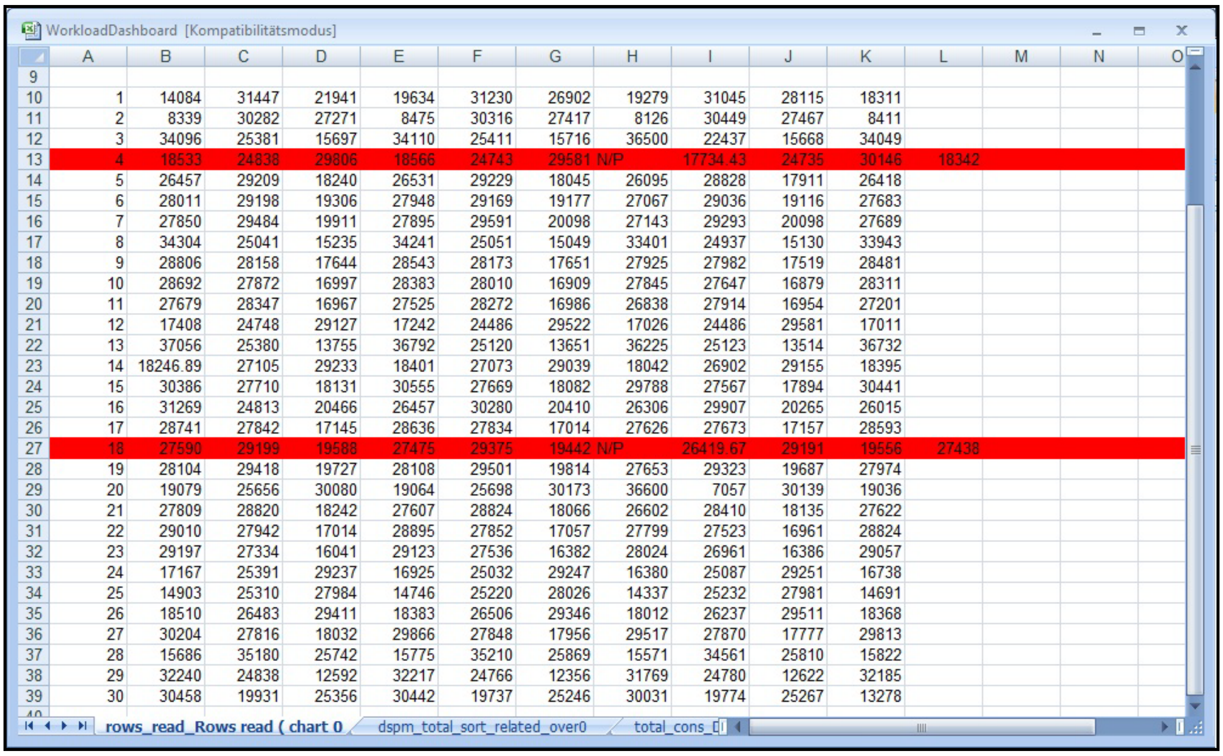
It can be seen in the diagram that the runtime of a small workload is too short to reach a *stable phase*. A stable phase can be understood in the way that once reached a certain point in time of the workload execution the metric is mainly influenced by the actual workload. The bufferpools are filled and the preparation phase of the workload scenario is not monitored anymore.

In order to deal with this behavior a new parameter “lastMinutes” is implemented. This parameter allows the user to set the time period for which he wants to collect data with the test environment. For example setting “lastMinutes” to 10 means that data monitored by OPM of the last ten minutes will be fetched and analyzed. Using the parameter “lastMinutes” for “TIME_SERIES” means to have exactly the same number of values for each iteration.

In some cases OPM does not receive data for a single time interval and returns the value “N/P”. The expected data is received a time interval later and causes this iteration to have one more value because all following values are also received a time interval later.

An example can be seen in Figure 4.2.2. It shows an extract of the result file of the workload dashboard. The pictured metric is counting the number of rows which have been read each minute. These values are in the columns B to K, respectively L for the red marked rows. In column A the number of the iteration is saved. The two rows which are marked red contain one “N/P” value each.

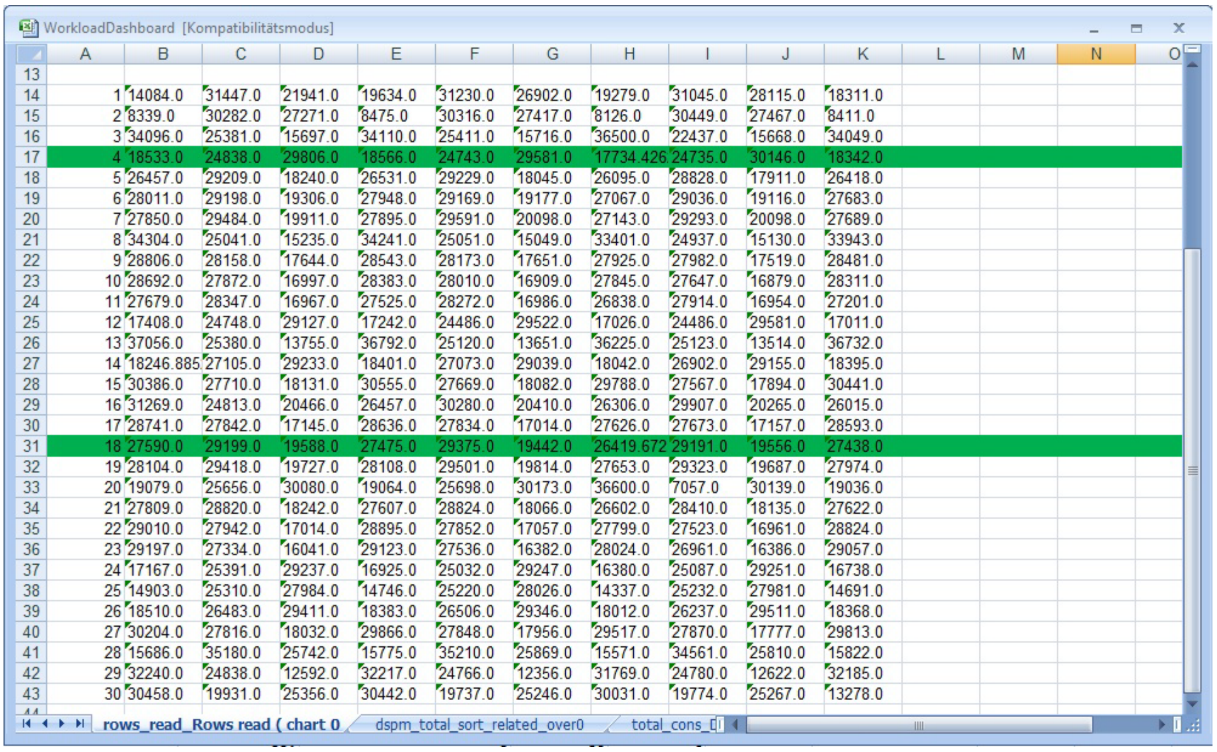
This occurrence leads to errors in the classification process. To avoid this, the data is normalized. If there is an “N/P” value for an iteration, this value is deleted and the remaining values are shifted. This results in equal numbers of values for each iteration. Figure 4.2.3 pictures the same metric after the normalization process. The rows which are marked green have been normalized.


WorkloadDashboard [Kompatibilitätsmodus]

	A	B	C	D	E	F	G	H	I	J	K	L
9												
10	1	14084	31447	21941	19634	31230	26902	19279	31045	28115	18311	
11	2	8339	30282	27271	8475	30316	27417	8126	30449	27467	8411	
12	3	34096	25381	15697	34110	25411	15716	36500	22437	15668	34049	
13	4	18533	24838	29806	18566	24743	29581	N/P	17734.43	24735	30146	18342
14	5	26457	29209	18240	26531	29229	18045	26095	28828	17911	26418	
15	6	28011	29198	19306	27948	29169	19177	27067	29036	19116	27683	
16	7	27850	29484	19911	27895	29591	20098	27143	29293	20098	27689	
17	8	34304	25041	15235	34241	25051	15049	33401	24937	15130	33943	
18	9	28806	28158	17644	28543	28173	17651	27925	27982	17519	28481	
19	10	28692	27872	16997	28383	28010	16909	27845	27647	16879	28311	
20	11	27679	28347	16967	27525	28272	16986	26838	27914	16954	27201	
21	12	17408	24748	29127	17242	24486	29522	17026	24486	29581	17011	
22	13	37056	25380	13755	36792	25120	13651	36225	25123	13514	36732	
23	14	18246.89	27105	29233	18401	27073	29039	18042	26902	29155	18395	
24	15	30386	27710	18131	30555	27669	18082	29788	27567	17894	30441	
25	16	31269	24813	20466	26457	30280	20410	26306	29907	20265	26015	
26	17	28741	27842	17145	28636	27834	17014	27626	27673	17157	28593	
27	18	27590	29199	19588	27475	29375	19442	N/P	26419.67	29191	19556	27438
28	19	28104	29418	19727	28108	29501	19814	27653	29323	19687	27974	
29	20	19079	25656	30080	19064	25698	30173	36600	7057	30139	19036	
30	21	27809	28820	18242	27607	28824	18066	26602	28410	18135	27622	
31	22	29010	27942	17014	28895	27852	17057	27799	27523	16961	28824	
32	23	29197	27334	16041	29123	27536	16382	28024	26961	16386	29057	
33	24	17167	25391	29237	16925	25032	29247	16380	25087	29251	16738	
34	25	14903	25310	27984	14746	25220	28026	14337	25232	27981	14691	
35	26	18510	26483	29411	18383	26506	29346	18012	26237	29511	18368	
36	27	30204	27816	18032	29866	27848	17956	29517	27870	17777	29813	
37	28	15686	35180	25742	15775	35210	25869	15571	34561	25810	15822	
38	29	32240	24838	12592	32217	24766	12356	31769	24780	12622	32185	
39	30	30458	19931	25356	30442	19737	25246	30031	19774	25267	13278	

rows_read_Rows read (chart 0 | dspm_total_sort_related_over0 | total_cons_

Figure 4.2.2: Data before Normalization Process


WorkloadDashboard [Kompatibilitätsmodus]

	A	B	C	D	E	F	G	H	I	J	K
13											
14	1	14084.0	31447.0	21941.0	19634.0	31230.0	26902.0	19279.0	31045.0	28115.0	18311.0
15	2	8339.0	30282.0	27271.0	8475.0	30316.0	27417.0	8126.0	30449.0	27467.0	8411.0
16	3	34096.0	25381.0	15697.0	34110.0	25411.0	15716.0	36500.0	22437.0	15668.0	34049.0
17	4	18533.0	24838.0	29806.0	18566.0	24743.0	29581.0	17734.426	24735.0	30146.0	18342.0
18	5	26457.0	29209.0	18240.0	26531.0	29229.0	18045.0	26095.0	28828.0	17911.0	26418.0
19	6	28011.0	29198.0	19306.0	27948.0	29169.0	19177.0	27067.0	29036.0	19116.0	27683.0
20	7	27850.0	29484.0	19911.0	27895.0	29591.0	20098.0	27143.0	29293.0	20098.0	27689.0
21	8	34304.0	25041.0	15235.0	34241.0	25051.0	15049.0	33401.0	24937.0	15130.0	33943.0
22	9	28806.0	28158.0	17644.0	28543.0	28173.0	17651.0	27925.0	27982.0	17519.0	28481.0
23	10	28692.0	27872.0	16997.0	28383.0	28010.0	16909.0	27845.0	27647.0	16879.0	28311.0
24	11	27679.0	28347.0	16967.0	27525.0	28272.0	16986.0	26838.0	27914.0	16954.0	27201.0
25	12	17408.0	24748.0	29127.0	17242.0	24486.0	29522.0	17026.0	24486.0	29581.0	17011.0
26	13	37056.0	25380.0	13755.0	36792.0	25120.0	13651.0	36225.0	25123.0	13514.0	36732.0
27	14	18246.885	27105.0	29233.0	18401.0	27073.0	29039.0	18042.0	26902.0	29155.0	18395.0
28	15	30386.0	27710.0	18131.0	30555.0	27669.0	18082.0	29788.0	27567.0	17894.0	30441.0
29	16	31269.0	24813.0	20466.0	26457.0	30280.0	20410.0	26306.0	29907.0	20265.0	26015.0
30	17	28741.0	27842.0	17145.0	28636.0	27834.0	17014.0	27626.0	27673.0	17157.0	28593.0
31	18	27590.0	29199.0	19588.0	27475.0	29375.0	19442.0	26419.672	29191.0	19556.0	27438.0
32	19	28104.0	29418.0	19727.0	28108.0	29501.0	19814.0	27653.0	29323.0	19687.0	27974.0
33	20	19079.0	25656.0	30080.0	19064.0	25698.0	30173.0	36600.0	7057.0	30139.0	19036.0
34	21	27809.0	28820.0	18242.0	27607.0	28824.0	18066.0	26602.0	28410.0	18135.0	27622.0
35	22	29010.0	27942.0	17014.0	28895.0	27852.0	17057.0	27799.0	27523.0	16961.0	28824.0
36	23	29197.0	27334.0	16041.0	29123.0	27536.0	16382.0	28024.0	26961.0	16386.0	29057.0
37	24	17167.0	25391.0	29237.0	16925.0	25032.0	29247.0	16380.0	25087.0	29251.0	16738.0
38	25	14903.0	25310.0	27984.0	14746.0	25220.0	28026.0	14337.0	25232.0	27981.0	14691.0
39	26	18510.0	26483.0	29411.0	18383.0	26506.0	29346.0	18012.0	26237.0	29511.0	18368.0
40	27	30204.0	27816.0	18032.0	29866.0	27848.0	17956.0	29517.0	27870.0	17777.0	29813.0
41	28	15686.0	35180.0	25742.0	15775.0	35210.0	25869.0	15571.0	34561.0	25810.0	15822.0
42	29	32240.0	24838.0	12592.0	32217.0	24766.0	12356.0	31769.0	24780.0	12622.0	32185.0
43	30	30458.0	19931.0	25356.0	30442.0	19737.0	25246.0	30031.0	19774.0	25267.0	13278.0

rows_read_Rows read (chart 0 | dspm_total_sort_related_over0 | total_cons_

Figure 4.2.3: Data after Normalization Process

The result files of the test environment are considered to be analyzed further for deriving correctness criteria. To have more information available for evaluation the result files are enhanced. Additional values are automatically computed and added to the result files, including the maximum deviation and the aggregation type of the metrics.

The next section contains the approaches and the performed tests to build the workload used for the classification process. This workload serves as a basis for the workload of the Semantic Validation Framework.

4.2.2 Workload used for the Classification Process

The workload used for classification has to meet several requirements. These requirements are described in Section 2.1.3. The implementation of the limited runtime can be done without problems. Though, easy adjustment and covering as many metrics as possible contradict. To find suitable workload, experiments with different scenarios are accomplished.

Tests with one scenario covering all metrics of the workload dashboard result in values which are not comprehensible. This is caused by the many different actions performed by the scenario. It includes deadlocks, hash joins, insert-, select-, update- and delete-statements as well as sorts and online analytical processing (OLAP) activities. The random factor in the order of execution ends in different values for different iterations. A comparison of different runs does not have a high significance.

This single but large scenario does not meet the requirement of easy adjustment. To change the amount of workload all types of activity have to be considered. It is hard to predict the impact of changes in the scenario because increasing the number of loops or changing the size of a table affects many metrics at a time.

Further testing is done with very easy adjustable scenarios. These scenarios execute only a single select-statement or causing one deadlock on the monitored database. In this case the impact of adjustments is predictable. For example increasing the loop by one should end up in two select-statements.

The coverage of these scenarios is very small. This means that for affecting all metrics many scenarios are needed. All of them are easy to adjust but only cover a special type of metrics. To have the same coverage as one run of the scenario for all metrics, a run of the test environment is needed for every easy adjustable scenario.

OPM itself causes activity on the database. It creates tables for all possible events. These tables are filled by the DB2 event monitors, which write data into the tables as soon as events occur. The created types of event monitors depend on the monitoring profile set by the user. Further, OPM executes SQL statements against the monitored database periodically. It uses DB2 facilities such as administrative views or table functions to retrieve the metrics and stores the values in the created tables. OPM reads across all tables every minute and stores the new data in its Performance Database. After the data is saved on the OPM side, it is deleted in the tables on the monitored database.

The amount of the activity caused by OPM is not negligible when validating the metrics. The count of read rows is between 1300 and 1600 rows per minute. There are 17 metrics on the workload dashboard affected by OPM. These metrics have values deviating around their average value. This means the impact of OPM could cause these metrics to be semi-deterministic or non-deterministic indifferent which workload is used.

These circumstances make easy adjustable scenarios mentioned above useless. The little amount of workload does not affect the metrics. Changes in this dimension are not visible because of the high activity of OPM. Workload with a certain intensity has to be executed to lower the impact of OPM and to be able to see results of the actual workload.

The most promising tests are done with two scenarios. Together these scenarios cover all metrics of the workload dashboard. The first scenario *olap_deadlock* executes OLAP functions and causes deadlocks. The second scenario *sort_hash* performs hash joins and sorts. These scenarios are chosen for the classification because the results are very stable. This means that very few metrics are changing their categories in different runs. Further, these scenarios can be adjusted with a manageable effort and the impacts of changes can be predicted with appropriate accuracy.

In the next section the range for semi-deterministic metrics is determined and the results of the classification using these scenarios are described.

4.2.3 Classification

Before the classification can be done, the range for semi-deterministic metrics is determined. Based on discussions and an analysis of the range distribution of metrics (IO dashboard and workload dashboard), the semi-deterministic range is set to 10%. A big part of the metrics which are not deterministic deviates up to 10% of their average value for every test run. There are less metrics deviating between 10% and 60%. The biggest part of the metrics has a deviation for more than 70%. In Figure 4.2.4 this behavior is pictured. If the value of the deviation is more than 100% it has been set to 100% automatically. The metrics are numbered consecutively. Every data point in the diagram is one metric. The y-axis stands for the deviation of the metric of its average value.

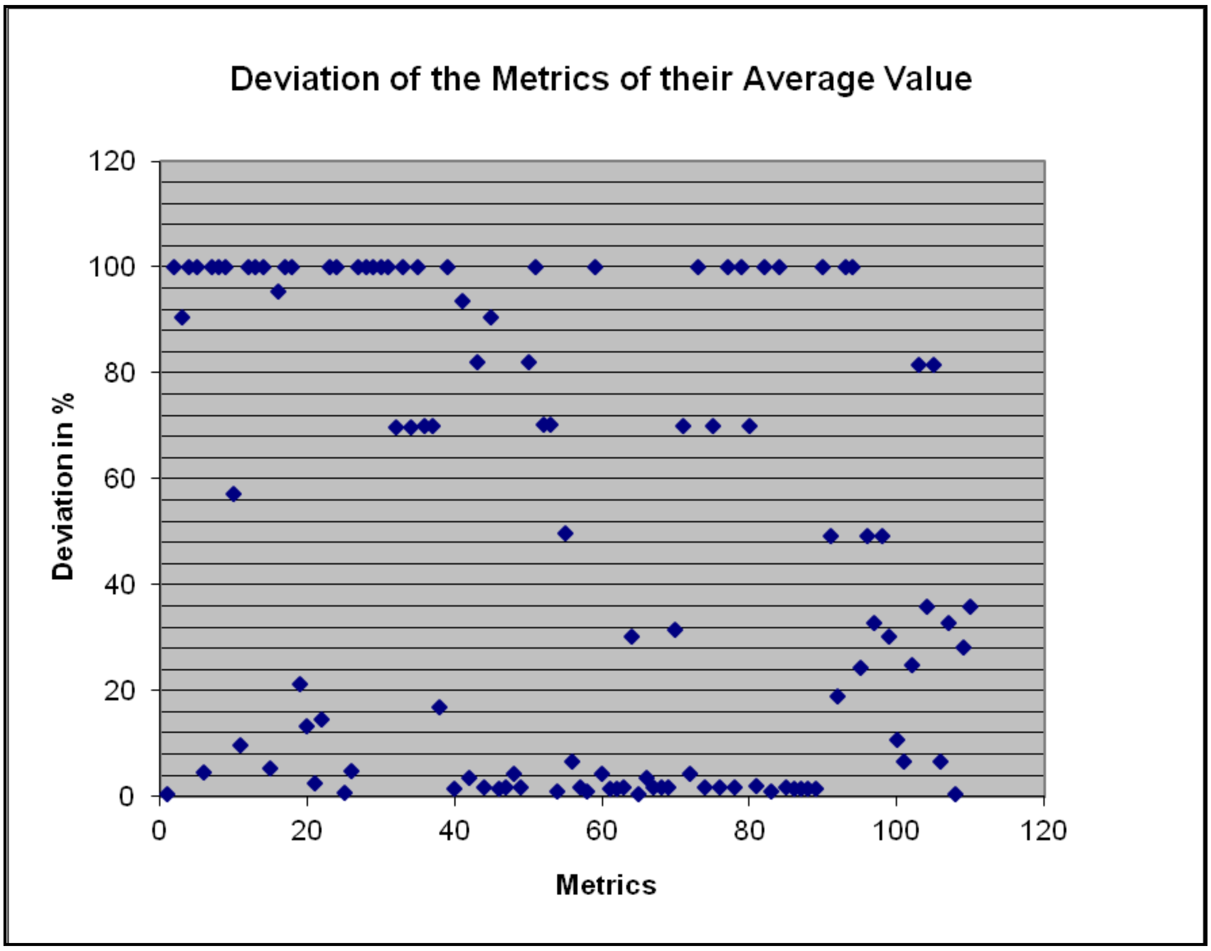


Figure 4.2.4: Deviation of the Metrics of IO and Workload Dashboard

If the semi-deterministic range is set to a higher value only few more metrics become semi-deterministic. For these metrics the correctness proving becomes imprecise because semi-deterministic metrics are validated different than non-deterministic metrics. Their validation is not suitable for highly deviating values. This means that the advantages of easier testing for semi-deterministic metrics do not prevail the disadvantages of the impreciseness of the semantic validation caused by increasing the semi-deterministic range.

For the classification each scenario, *sort_hash* and *olap_deadlock*, is used in seven runs with 30 iterations. 30 iterations are chosen because it is a “common rule of thumb” to lower the impact of possible outliers [27]. One run takes about 12 hours and is executed on a server during the night to decrease the influence of other people working on the same machine.

In these seven runs of each scenario only two metrics of the workload dashboard changed their category. All other metrics are classified in the same category for every run.

One of the metrics, which are changing categories, is counting the maximum number of coordinator agents working at the same time. A coordinator agent is requested if an application is connecting to a database or instance [28]. This metric is classified either deterministic or semi-deterministic. Based on discussions with the team, which is responsible for the testing of OPM, this metric is classified manually as semi-deterministic. The main reason for this decision is the easier correctness proving. Changing the category from semi-deterministic to deterministic is easier to validate. If a semi-deterministic metric becomes deterministic in a test, it is not marked as wrong automatically. Its values are all the same and the deviation of the expected value is 0. This deviation is surely within the semi-deterministic range.

The second metric which is changing categories during the different runs is ratio of the created agents vs. the assigned agents from the pool. An agent is a process which is responsible to execute the request of a client application. In contrast to the metric counting the number of coordinator agent this metric is regarding all types of agents (coordinator agents, subagents, associated agents, primed agents). The number of created agents is determined by the parameters of the DB2 instance [29]. This metric is switching categories between semi-deterministic and non-deterministic.

The same argument as mentioned above is true for this metric. It is easier to validate the metric if it is categorized as non-deterministic. If the metric would be classified as semi-deterministic it is marked as incorrect in every case the values are deviating more than the predefined range.

The classification of the metrics of the workload dashboard resulted in six deterministic metrics, two semi-deterministic metrics and 32 non-deterministic metrics. The next section describes the different processes of semantic validation for each metric category.

4.3 Process of Semantic Validation

In this section the semantic validation process is described. As mentioned above this process differs depending on the classification of the metric. The procedure for each metric category is explained in the following.

4.3.1 Semantic Validation Process of Deterministic Metrics

The semantic validation of deterministic metrics consists of two major steps. First it is checked if the category matches. This means that the result of the test run for each iteration has to be the same – the category has to be deterministic. The next step is to verify constraints. For example it is checked if DB2 parameters are within the allowed range. Further, there are values which indicate that there could be a defect in OPM. These values are “NULL”, “NOT_RETURNED”, “N/P” and in some cases “0”. Their occurrence has to be caught and reported to the user.

The following table contains all deterministic metrics of the workload dashboard. It further includes a description and the constraints of each metric.

Metric	Description	Constraints
DBC6	Maximum number of concurrent applications connected to the database	-
max_connections	Maximum number of applications allowed to connect to the database	1 – 64 000
sortheap	Maximum number of pages of the private / shared memory available for private / shared sorts (4 KB per page)	16 – 4 194 303
sheapthres_shr	Threshold for the size of the shared database memory used for sorts (4 KB per page)	250 – 2 147 483 647
sheapthres	Soft limit for the amount of memory used for private sorts (after this limit the provided memory for sorts is reduced)	0, 250 – 2 147 483 647
DBMC501	Maximum number of coordinator agents	0 – 64 000

Table 4.3.1: Deterministic Metrics

Some of the DB2 parameter, which influence these metrics, can be set to "automatic" and DB2 is looking for the best value. To improve and to facilitate the validation process the tester should set those parameters to a stable value manually. In this way assertions about the correctness can be drawn definitely. The value of the parameter is saved in the framework together with the corresponding metric and is compared to the actual result of the test run.

4.3.2 Semantic Validation Process of Semi-Deterministic Metrics

The validation process for metrics of the category semi-deterministic starts with the category verification. If a semi-deterministic classified metric becomes non-deterministic in the test run it is considered to be incorrect and the user is notified. There is the possibility that the metric becomes deterministic and can be considered as correct.

The difference between deterministic metrics and semi-deterministic metrics is the size of the deviation of the average value. For deterministic metrics this deviation is 0% and for semi-deterministic it is within 10%. This means that metrics which have been classified semi-deterministic during the classification process can have deterministic values for test runs and still be correct because a deviation of 0% does not contradict the rules for semi-determinism in this case a priori.

For example, during classification some metrics switched between the categories deterministic and semi-deterministic for different runs of the test environment. These metrics have been classified as semi-deterministic because they did not have the same values for each run and each iteration which is the main characteristic of deterministic metrics.

In case the resulting category of the tested metric is deterministic, the framework checks if the values themselves do indicate an OPM defect. These values are the same as for the deterministic metrics: “NULL”, “NOT_RETURNED”, “N/P” and for some metrics “0”. Depending on the workload scenario, the result of metrics which are not affected can adopt those values. These metrics are ignored for validation in this run and a different scenario has to be used. After the metrics passed this test all constraints are verified. For example, these can be thresholds derived from previous testing or DB2 parameters.

The Table 4.3.2 contains the two semi-deterministic metrics of the workload dashboard, the description of the metrics and the constraints set by the DB2 database system.

Metric	Description	Constraints
DB2390	Number of registered coordinator agents and subagents in the database manager instance	defined by parameter “maxagents”
coord_agents_top	Maximum Number of concurrent coordinator agents	defined by parameter “max_coordagents”

Table 4.3.2: Semi-Deterministic Metrics

4.3.3 Semantic Validation Process of Non-Deterministic Metrics

In the first step of the Semantic Validation of non-deterministic metrics it is checked if the category determined in the test run is non-deterministic or semi-deterministic. If the metric is categorized deterministic, this may indicate an error and further investigation has to be done. The next step is to verify the constraints for every metric. Equal to the other metrics categories these constraints can be affiliated to DB2 limitations in most of the cases.

The constraints checked in step two are vague. For example, within a range of 250 to 2 147 483 647 many wrong values or patterns can occur. This means that for non-deterministic metrics further validation has to be done to prove the correctness. It is not sufficient that the values of the metric are within the range allowed by DB2 and that the metric is categorized as non-deterministic.

In the next chapter, the approaches introduced in Chapter 3 are revisited. It is checked whether they fit to prove the semantic correctness of non-deterministic metrics. Further, it is evaluated whether they are applicable under the given circumstances.

5 Validation Approach for Non-Deterministic Metrics

In the first part of this chapter the approaches of Chapter 3 are revisited. They are examined considering the criteria for the Semantic Validation Framework. It is checked which approaches are suitable for semantic validation in case of non-deterministic OPM metrics. The second part of the chapter describes the chosen solution in more detail. In the third part the chosen solution is transferred to OPM.

5.1 Approaches – Revisited

In this section approaches and techniques introduced in Chapter 3 are examined to evaluate if they would be suitable to be implemented in the Semantic Validation Framework. The result of the evaluation for each approach depends on several criteria.

- **Realization**: The approach has to be realizable using the available test environment. It has to be possible to build a prototype in the given timeframe.
- **Automation**: The approach should not require user interaction after starting the test.
- **Maintenance**: The approach has to be maintainable in an easy way. This includes adding new metrics and discarding unused metrics.
- **Robustness / Solidity**: The approach should be robust against changes made in OPM during development. This includes changes in the Repository Server, the Performance Database and the function accessing the data.

The next sections describe how far the approaches meet these requirements.

5.1.1 Static Analysis – Revisited

In Chapter 3 two different techniques of static analysis are introduced, model checking and theorem proving. In model checking a model of the system under test is built. The specification has to be formalized into properties, which the model has to fulfill. A model checker verifies whether the model conforms to its specification.

Model checking validates a very large set of possible execution paths of the software and it gives a counter example for every defect found. These are two great benefits of model checking. One of the disadvantages is the state space explosion problem.

Choosing model checking for the Semantic Validation Framework provides large coverage and automatic defect localization. Automation of the testing is possible. However, modeling OPM to a formal description is not practicable in a short time period. The model has to be built from scratch, examining hundreds of thousands of lines of code.

For every new metric or old metric, which is not used anymore, the model and the properties have to be adjusted. The same is true for changes in other parts of OPM. This leads to a complicated and time consuming adjustment process, every time changes are performed. These adjustments should be prevented in the Semantic Validation Framework (Section 2.2.1).

Theorem proving is a method based on mathematics to check if functions are corresponding to their specification. It is tested if certain preconditions and postconditions are fulfilled. During this test the software is not executed but axioms and inference rules according to the behavior of the software are used to derive results.

Automation is possible in case of theorem proving. The implementation has to choose the fitting rules or axioms and use them on detected preconditions. If the results are the corresponding postconditions the function is correct. The handling of new and old metrics is practicable. For new metrics conditions to check have to be found. For old metrics the testing is not done anymore.

Changes in OPM influence theorem proving. The inference rules have to be checked if they are still valid and maybe new rules have to be developed. Similar to model checking this means that for every new version of OPM the framework has to be validated and adjusted again.

To use theorem proving proper pre- and postconditions have to be figured out. There is an infinite number of possible inputs and parameters for OPM. For example, every different workload combined with database settings, as the size of sort heaps. Everything has to be considered when evolving these conditions. In this case, achieving exact pairs of pre- and postconditions is not practicable. These shortcomings mean that theorem proving is not suitable for this project.

Both static analysis techniques suffer from the large effort they cause if changes in the software under test are done. Even if using these techniques makes the results more meaningful, they are not fitting in this case.

5.1.2 Dynamic Analysis – revisited

Runtime assertion checking and testing are described in Chapter 3 as two different approaches of dynamic analysis methods. In runtime assertion checking requirements of system states are checked during runtime. These requirements can be Boolean expressions or constraints. A function is correct if all assertions are fulfilled. It is possible to implement runtime assertion checking directly in the code, which makes it easier to detect the location of errors.

It is possible to automate runtime assertion checking. During execution of the software it is checked automatically if the actual state satisfies the given assertions. The results are messages which assertions are fulfilled and which are violated.

Related to both approaches of static analysis the realization is very time consuming. The specification and the code have to be examined to derive valid assertions. Further, for every change in the code it has to be checked if assertions have been affected. For adding new metrics the examination has to be done again. This leads to a large effort in maintaining the framework. These are characteristics, which are not suitable for the Semantic Validation Framework.

The second approach is testing. Testing is the execution of the software under test under certain circumstances. These circumstances are called test cases. To verify if the software is implemented correctly an oracle has to exist. An oracle is a method to decide whether a test case succeeded or not. Testing methods differ by the way they deal with the code, how test cases are chosen and what foreknowledge of the output is available.

It is possible to realize testing when considering the available environment. Test cases consist of workload and the environment of the monitored database and OPM. The results can be processed further with the test environment. Logics which store this data in result files are implemented already in the test environment. To evaluate these results, functions which check if correctness criteria are fulfilled have to be developed.

Once, the test is started the test environment does not require user interaction anymore. The handling of new metrics can be done more easily than with static methods or runtime assertion checking. Corresponding workload and correctness criteria have to be developed, without having to examine the code of OPM.

The same is true for changes in other parts of OPM. Using black box testing the code itself is not taken into account. This means that no adjustments have to be done if the way metrics are computed has changed for example. The results of the tests are not affected because they consider the input and the output. These properties of testing allow to maintain the Semantic Validation Framework with reasonable effort.

Further approaches, for example the S-Match algorithm [26], which compute semantic relations are not suitable for the framework. They are able to check if two inputs correspond semantically to each other but they cannot test the correctness of the inputs.

Table 5.1.1 contains the summary of the revision of the approaches. For every approach it is marked with “+” that it meets the criterion or with “-“ that it does not meet the criterion.

Approach	Realization	Automation	Maintenance	Robustness / Solidity
Model Checking	-	+	-	-
Theorem Proving	-	+	+	-
Runtime Assertion Checking	-	+	-	-
Testing	+	+	+	+

Table 5.1.1: Summary of the Evaluation of the Validation Approaches

Automation is possible for every approach but only theorem proving and testing are maintainable with affordable effort. The requirements for the Semantic Validation Framework and the feasibility of the approaches mentioned above lead to the conclusion that testing is the most suitable approach. This approach is the only one which is realizable and robust in the case of the given circumstances. The next section contains detailed information about the testing approaches used for the framework.

5.2 Chosen Solution – Testing

The complexity of correctness criteria for testing ranges from simple for deterministic metrics to very complicated for non-deterministic metrics. For metrics of the category deterministic few more knowledge is needed. For example the number of processed statements cannot be negative or a percentage value has to be in the range between 0 and 100. The appropriate function is considered to be correct if the value of the metric fulfills these criteria and is the same for each iteration of the framework. This can be checked in an easy way by comparing the values.

Metrics of the category semi-deterministic are handled in a similar way. The difference between the two validation processes is that these metrics do not have the same value for each iteration. The values are located within a certain deviation of the average value. This correctness criterion can be validated by computing the average and comparing it with every value.

For non-deterministic metrics none of these correctness criteria is suitable. This means that more complex test cases and criteria for correctness have to be developed. For these metrics of OPM a variation of statistical testing is used. In the following sections the basic theory is explained because the simple approach of statistical testing described in Chapter 3 has to be adapted to fit for OPM.

5.2.1 Statistical Hypothesis Testing

The problem of testing randomized software (in this case non-deterministic metrics) is that for the same input different output is produced. Thus testing if the values for a specific input are the same or are within a certain range for each iteration is not possible. However, an oracle to test if an output is correct for the given input is needed to draw conclusions about the correctness of the software.

Statistical hypothesis testing is based on checking the distribution of the output values. There are different methods to verify the output depending on the environment.

- If theoretical output values are known the expected distribution can be computed and the real output is checked against this distribution.
- If there is a reference implementation available it is possible to verify against the corresponding output of this reference.
- If neither theoretical values nor references exist, an expected distribution can be derived based on the central limit theorem. It implies that if stochastic variables are independent, the variance of their distribution is finite, and the sample size is high enough, the centered and scaled mean is normally distributed. In this case, two more parameters are needed to center and scale the mean - the true mean and the true standard deviation [27]. These parameters are not known in most cases and have to be estimated.

Statistical hypothesis testing provides a technique to verify randomized software to a certain probability. It is not possible to make sure that there are no more defects. It may happen that correct values are recognized as wrong and wrong values as correct because the correctness criteria are not exact [22] [23].

5.2.2 Metamorphic Testing

First, metamorphic testing is a technique to build so called “follow-up” test cases based on previous successful test cases. A test case is considered to be successful if the output is correct. In some cases this criterion can be softened if the follow-up test cases can prove the correctness of the output of the successful test case. Here successful means that at least an output is available. The idea is that these follow-up test cases are related to the successful ones. This leads to a relation between the results and this relation can be checked without having an exact oracle [23].

An extension of metamorphic testing is to use the meaning of the functions under test to find correctness criteria. From the meaning characteristics are deduced which are necessary for the correctness of the functions. Considering these characteristics, related test cases can be derived and their results can be compared [30].

A very simple example for the extension of metamorphic testing can be seen in Figure 5.2.1. A function computing the product of two numbers is tested. The first test case T1 consists of the numbers 2 and 3 and the result is 6. At this point of time the correctness of the result is unknown. The second test case T2 is the result of T1 and 1/3 which is the inverse of the second number of T1. The expected result is the first number of the first test case 2.

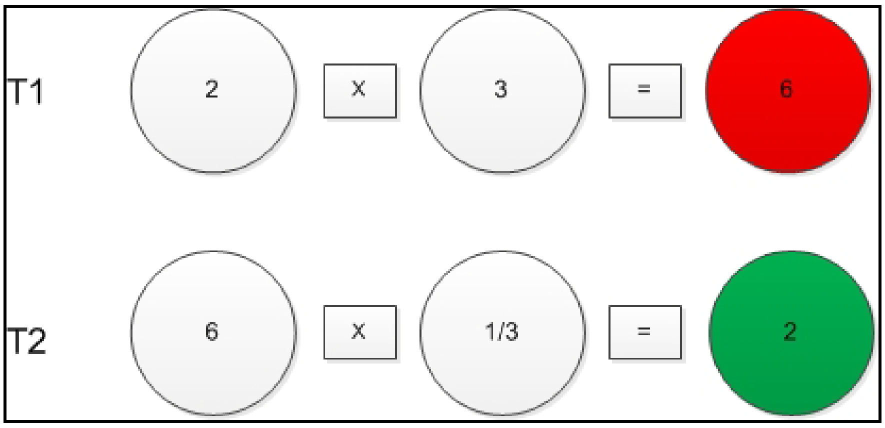


Figure 5.2.1: Example for Metamorphic Testing

Knowing about the mathematical relations between these three numbers does not require to know that 6 is the correct result of the first test case. In this example, metamorphic testing allows to draw conclusions about the correctness of functions at least for the given test cases. To gain more confidence about the correctness more test cases and the corresponding follow-up test cases have to be verified.

In the same way as other testing methods, metamorphic testing is only able to detect errors but not able to prove the absence of errors. Requirement for metamorphic testing is the knowledge of metamorphic properties of the tested software. In the example in Figure 5.2.1 the mathematical relation is the metamorphic property. In case of OPM, the metamorphic properties are the different behaviors of metrics when changing workload. These properties can be derived from the specification of the metrics. To derive meaningful properties expert knowledge about DB2 is essential.

5.2.3 Statistical Metamorphic Testing

Applying statistical hypothesis tests requires at least some knowledge about the distribution of the output or a reference to check against. This reference and knowledge are often not available, especially in the case of OPM. Using metamorphic testing leads to similar test cases and coverage is a problem. For OPM it is hard to find suitable metamorphic properties to derive follow-up test cases.

A solution for these shortcomings is statistical metamorphic testing, a combination of the two approaches mentioned above [31]. In statistical metamorphic testing the software is executed several times with different input parameters. This input is correlated among each other. For example, the input can vary over the number of executed statements. This means, that the different workloads distinguish in the number of statements which are executed during the runtime. The idea is that if there is a relation between the input, there should be a relation between the output.

The output relation can be derived from the relation of the input. This makes it possible to verify the different outputs by checking if they fulfill the derived relation. In other words, this allows to perform statistical hypothesis tests because a theoretical distribution based on the input is available.

The whole process of statistical metamorphic testing is as follows:

1. A suitable relation has to be found and corresponding input has to be created.
2. The software has to be executed using the input of step one.
3. For the output generated in step two statistical hypothesis tests have to be performed to check if the relation is fulfilled.

To gain more confidence and to minimize the influence of the environment, step two and three are performed several times. The software is considered to be correct if it passes the majority of the statistical hypothesis tests in step three.

For statistical metamorphic testing knowledge about the meaning of the tested functions should be available to be able to find suitable input. Besides that, there are no more requirements. The correctness criteria are deduced from the meaning of the function and the different inputs.

Transferred to OPM the input are the SQL statements executed by the workload tool. The execution of OPM is done by the test environment. The number of executions can be defined by the user. Proper relations of the metrics are deduced from the semantic of each metric and the used workload. Statistical hypothesis tests check the values of metrics whether they fulfill the deduced relations.

This approach allows to build a reference for faster and easier tests. Once a metric is verified, the produced output or derived patterns can be used to check newer versions. Under the same preconditions, including environment and workload, the metric should have nearly the same output values as the reference.

Considering influences, for example, different load factor of the server, and the random character of the metrics it is impossible to receive exactly the same values. In case of reference the correctness criterion is not equality. A metric can be considered as correct if the majority of the output values is within a determined maximum deviation of the reference.

In Section 5.3 statistical metamorphic testing is transferred to OPM. It is described how correctness criteria are derived, considering the workload and the values of the metrics.

5.3 Statistical Metamorphic Testing in Case of OPM

To be able to draw conclusions about the semantic correctness of metrics statistical metamorphic testing has to be adapted to OPM. For OPM, the process of semantic validation of the non-deterministic metrics includes the following steps:

1. Workload is created and correctness criteria are derived concerning the workload, the output and the semantic of the metrics.
2. OPM is executed using the input / workload of step one.
3. For the output generated in step two, statistical hypothesis tests have to be performed to check if the correctness criteria are fulfilled.

The first step for statistical metamorphic testing is creating workload. This workload has to be customizable to allow the user to create different test runs using different but related workload. The related workload is used to derive relations between the results themselves and between the workload and the results. These relations serve as correctness criteria for the statistical hypothesis tests. A theoretical example of possible relations between the workload and the resulting values can be seen in Figure 5.3.1.

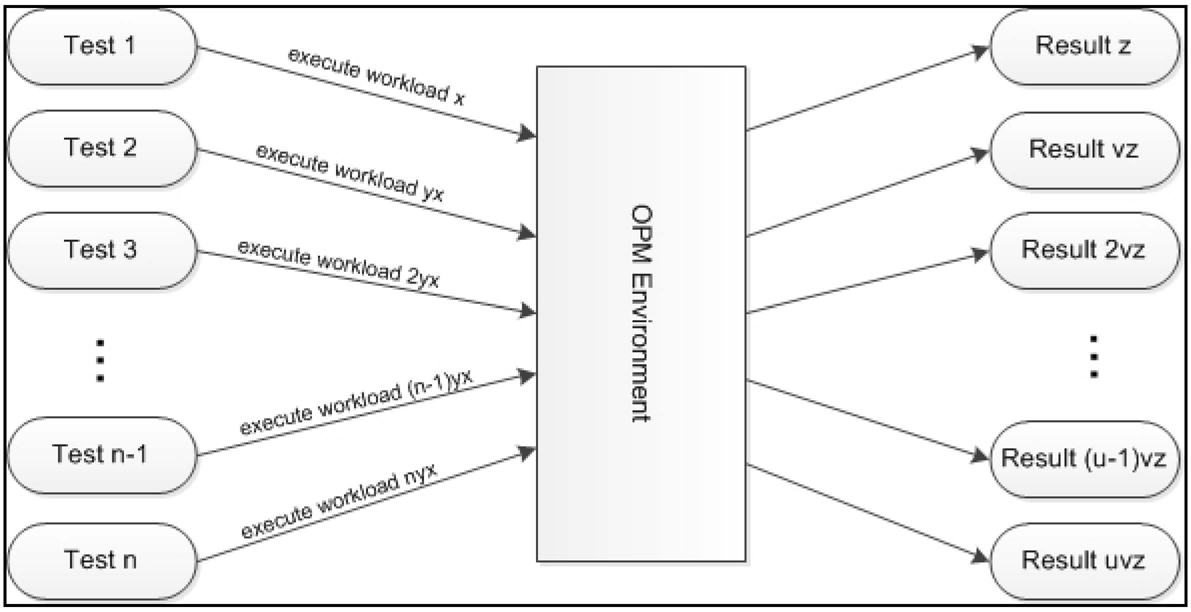


Figure 5.3.1: Related Workload – Related Results

The “OPM Environment” consists of OPM, DB2 and the monitored database. During the first test run workload *x* is executed on the monitored database. The result of this run is *z*. In the second run workload *yx* is performed on the database. Workload *yx* means that there has been a modification *y* to the first run. Accordingly, 2yx means that the modification is done twice. This modification can include changes in the amount of the workload or changes in the performed SQL statements. Further it is possible to make modifications in the workload preparation phase, for example changing the size of tables. Another option is to modify DB2 parameters.

The resulting values of the second test run are expected to be related to the results of the first run. This is illustrated by the *v* in “Result *vz*”. This expectation is based on the fact that both workloads are related and this should lead to related results.

For each test run, workload is built by making changes to the workload of another run. Every workload can be built from every other workload by modifying certain parameters. This leads to the conclusion that every result can be computed by using any other result and modifying it with the right parameters. For example, result *2vz* can be computed by doing the modification which has been done to derive result *vz* from result *z* twice.

To be able to draw conclusions about the correct behavior of the output values, a validation of the derived relations is done. For this validation two things have to be considered. The first is the semantics of the tested metric. The second is the relation between the several inputs.

Knowing about the meaning of the metrics, it is possible to predict the behavior of its values for different workload. This prediction is limited. With the knowledge of the meaning it is only possible to make assumption about how the values will change. Additionally, knowing about the relation between the workloads, predictions can be made more precisely. This means that it is possible to make assumptions about the proportion the values will change.

The following example is based on a metric which is counting the number of processed SQL statements. Increasing the number of statements in the workload means a higher value for this metric. This is the assertion which can be drawn when knowing the semantics. If the quantity *x* of the number of statements the workload has been increased is known, drawing more detailed conclusions about the behavior of this metric is possible. Its value increases at least for the quantity *x*. Remark that the influences of the environment do not allow to predict the exact results computed by OPM.

A validated relation is required to predict results for every input which are correct to a certain degree. This is a precondition for step three where the results generated by running the tests on OPM are examined. It is checked if the derived correctness criteria of the metrics are true for the output values.

Step two is to run tests with the workload created in the previous step. For one complete validation the tests are performed on the same environment. The environment influences the results of OPM. The database system DB2 and OPM itself are executing SQL statements and are making adaptations and improvements during the test runs. These events are monitored by OPM in many cases and flow in the result values. Performing each test run under the same conditions makes it easier to predict the behavior of the environment. This allows to filter out the actions of DB2 and OPM to a certain degree.

Filtering out all environmental influences is not possible. All non-deterministic metrics of the workload dashboard are TIME_SERIES. This means that for every minute OPM collects one value. Since, the actions of the environment are not the same for each one minute time interval these values contain different amount of “environmental workload”.

In step three the results generated in step two are examined. It is checked if these results fulfill the criteria developed in step one. The setup for the semantic validation process is shown in Figure 5.3.2.

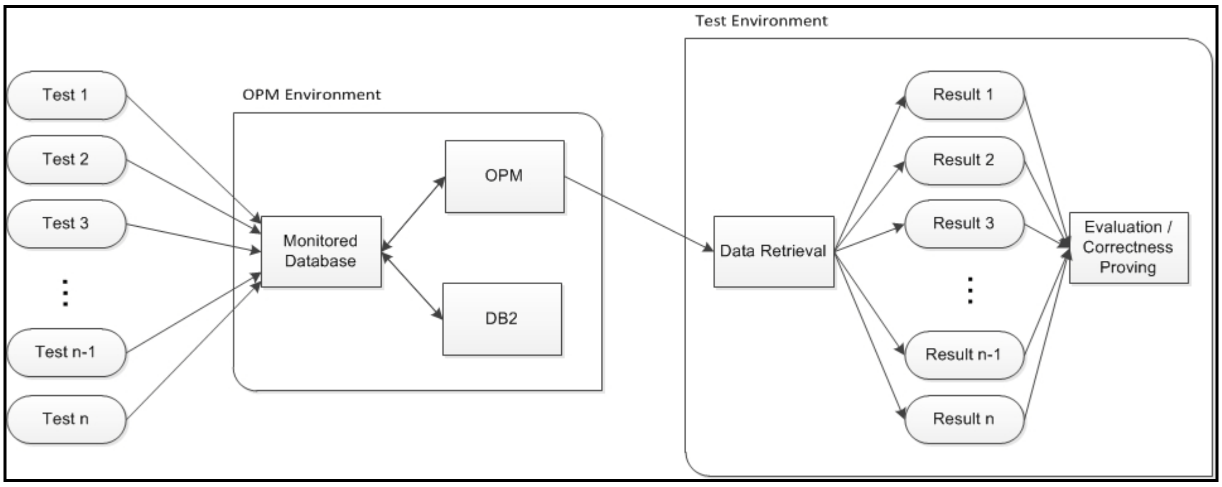


Figure 5.3.2: Setup for Semantic Validation

Several tests with related workload are executed on the monitored database. OPM and DB2 influence the created data, which is collected and stored by OPM. The test environment retrieves this data and creates result files to save it. These files are evaluated using the derived and validated relations between the results themselves and between the results and the workload. Each test in Figure 5.3.2 ends in one result. For each result statistical hypothesis tests are performed. A metric is considered to be correct if it passes the majority of these tests. The result of this process is a statement about the correctness for each tested metric.

In Chapter 6 the theory of statistical metamorphic testing of the non-deterministic metrics is transferred to the praxis. For two metrics the whole validation process is performed. The derived correctness criteria are implemented in a prototype, which is able to validate these metrics in future testing.

6 Prototype for Statistical Metamorphic Testing

The semantic validation of the OPM metrics differs depending on the metric category. For non-deterministic metrics statistical metamorphic testing is used. The goal of this chapter is to introduce a prototype, which validates two performance metrics of the workload dashboard. Before the prototype can be implemented in the test environment, correctness criteria for both metrics have to be derived and validated. This is done in Section 6.1. Section 6.2 contains the prototype.

6.1 Statistical Metamorphic Testing transferred to the Praxis

This section explains the procedure of semantic validation of OPM based on two metrics. Steps one to three of Section 5.3 are performed: workload is created, correctness criteria are derived and validated and statistical hypothesis tests are done. The goal is to develop criteria, which are always valid if the metric is semantically correct.

The metrics for which the semantic validation process is described are the metric DBSE427 and the metric sort overflows. The metric DBSE427 is the total number of processed sorts. The metric sort overflows is providing the ratio of sorts which caused overflows to the total number of sorts. This metric is a percentage value. Both metrics are of the type “TIME_SERIES”, which means that they have values for every one minute time interval.

DBSE427 is chosen because it is possible to predict the number of sorts performed by the used workload scenario very exactly. Further, the behavior of DBSE427 influences many metrics of the workload dashboard. This means that if this metric is correct, the validation of the influenced metrics can be done on a strong basis. One of these metrics is sort overflows. Sort overflows is validated using DBSE427 after it is proved that its behavior is semantically correct.

The section is divided in three parts. In part one, the parameters which influence the metrics, the used workload and the general testing process are described. Parts two and three explain the details for each metric.

6.1.1 Used Workload and Testing Process for both Metrics

To build proper workload, which creates data for both metrics, it is necessary to know how to manipulate their values. DBSE427 is only influenced by the number of sorts executed per minute. The metric sort overflows depends on the total number of sorts, sorts which causes overflows, and indirectly on the size of the sort heap. The size of the sort heap determines the memory which is available for each sort. This means, there are two possibilities for the sorts. Either the provided memory is enough for the sort which does not cause overflows or the memory is too small and the same sort causes an overflow every time.

To influence the behavior of these two metrics the number of sorts per minute can be varied. Further, the size of the sorts can be modified to have different numbers of sorts which cause overflows and which do not cause overflows. This includes the size of the tables from which the data for the sorts is selected. If there is less data to sort less memory is needed. Changing the DB2 parameter *sortheap* in the monitored database configuration is another possibility to influence sort overflows.

The execution time for one test run is between 12 and 15 hours. For each modification several test runs are performed to reduce the influence of outliers. This means, for this thesis there is not enough time to run tests for all possible parameters. For the validating DBSE427 the number of sorts per minute is modified. For sort overflows, additionally, the size of the parameter sortheap is changed and different kinds of sorts are used.

To be able to draw assertions about the correctness of DBSE427 as exact as possible the used workload scenario *semantic_sort_ms* creates a predictable number of sorts per minute. This leads to the possibility to make assumptions about the behavior of other metrics influenced by the number of sorts based on very exact values. To allow the user to adjust the workload, the scenario provides two integer parameters as scenario arguments to modify the number of sorts per minute. These arguments can be seen in Figure 6.1.1.

```
<requiredargs>
  <arg  default="350"  description="Number  of  loops"  key="loops"
type="int"/>
  <arg  default="2500"  description="Waiting  time  in  each  loop"
  key="loopwaittime" type="int"/>
</requiredargs>
```

Figure 6.1.1: semantic_sort_ms Arguments

The first one is the number of loops. The loop contains the SQL statements which select data from the monitored database and sort this data (see Figure 6.1.2). The second parameter is the waiting time which allows the user to adjust the runtime of one loop. This is needed to control how many sorts are performed during a one minute time interval. To validate sort overflows the scenario is adapted which is explained in Section 6.1.3.

```
<loop cycles="int:use:scenarg:loops">
 <echo message="int:use:var:loop_count"/>
  <!-- performing joins -->
  <sql>
      <string value="SELECT h1, h11 from table1 INNER JOIN table2
      ON table1.h11 = table2.h21 ORDER BY h11, h1 ASC"/>
  </sql>
  <sql>
      <string value="SELECT h2, h21 from table2 INNER JOIN table1
      ON table2.h21 = table1.h11 ORDER BY h21, h2 ASC"/>
  </sql>
  <waittime millis="int:use:scenarg:loopwaittime"/>
  <setvar key="loop_count" value="int:plus:1"/>
</loop>
```

Figure 6.1.2: Main Loop of the semantic_sort_ms Performing the Sorts

To be able to compare different output it is necessary to change the workload activity per minute because both metrics are TIME_SERIES. This is not possible by modifying only one parameter.

For example, if the number of loops is increased but the waiting time is retained the overall runtime of the workload is increased, too. The number of sorts per minute stays the same because the runtime of one loop does not change. Only a certain number of loops can be processed each minute based on the runtime of one loop.

Every test run consists of 30 iterations using the scenario *semantic_sort_ms*. The runtime of this scenario is about 15 minutes. After the scenario is finished, the test environment captures the data for the last ten minutes. In this case the data of the preparation phase is not collected. There is enough time for the warm-up phase which is not included in the last ten minutes. This process produces result files with 300 values for both metrics, ten for each iteration.

It is necessary to run a sufficient number of tests to be able to draw correct assertions about the behavior of metrics. This number differs depending on the metric and has to be determined for each case. If the number of test runs is too small the possibility exists that the behavior is interpreted wrongly or patterns such as asymptotic properties or bounds are missed.

In the following, every workload used for test runs is based on the "basic workload". This basic workload has 350 loops and 2500ms waiting time. Related workload is computed by using these parameters as a basis. For example, "50% workload" means 175 loops and 5000ms waiting time. This allows to compute a relation between each workload, which is essential to conclude to the behavior of the output.

The DB2 parameter "sortheap" is set to its minimum 16. This means there is memory of 64kb (16 * 4kb) available for every sort. Using the basic workload, 24 loops are performed during one minute. In each loop two SELECT statements are executed. Each statement triggers two sorts. The first one during the inner join, because DB2 is using a merge scan join, which requires sorted input [32]. The second sort is done to sort the selected values. This results in approximately 96 sorts per minute for the basic workload. The sort heap size is not sufficient for the sorts. Each sort of the scenario is causing an overflow.

This allows the user to determine the total number of sorts which are performed per minute. The knowledge about this number is used to validate the correctness of both metrics. DBSE427 can be checked directly. For sort overflows proper correctness criteria are developed regarding DBSE427 and further parameters which have influence on its behavior.

To simplify the validation, the average values of the 300 values for each run are computed and used for further investigation. This is possible because the average value is a sufficient approximation of the real behavior of the metric. To prove this assumption a frequency analysis is done for both metrics. The deviation in percent of the average value is computed for a sample size. Based on these values it is counted how many values are within a certain range around the average.

For the metric DBSE427 about 70% of all values are within 4% and 95% are within 8% deviation of the average value. Only 4% deviate for more than 10%. For sort overflows 77% of all values are within a 4% deviation of the average value. Around 19% deviate between 10% and 14%. The results of the frequency analysis reveal that the average value is proper to use. A high majority of the values are within 4% deviation.

In the next section, the metric DBSE427 is validated. Correctness criteria are derived and it is checked whether they are fulfilled.

6.1.2 Semantic Validation of DBSE427

To check the correctness of the metric DBSE427, the number of sorts performed by the workload scenario every minute is computed. As described above, the value for the basic workload should be around 96. This does not include sorts done by OPM. DBSE427 counts all sorts on the monitored database. The observed value is 20 sorts higher than it should be for the workload scenario.

To find out if this is caused by OPM another workload scenario *just_wait_ms* is executed and the results are examined. The scenario runs for 15 minutes and does not execute statements. The results of these test runs are the activity of OPM when there is no workload running on the monitored database. This analysis reveals that OPM is doing around 20 sorts per minute.

Owing to exterior influence of DB2, OPM, and the environment including the server among others, it is impossible to execute exactly the same number of sorts each minute. The expected value for DBSE427 is the number of sorts of the workload scenario plus the 20 sorts of OPM. The real values differ most often between 0 and 3 sorts of this expected value up to 200% workload.

This inaccuracy is most probably caused by the environment. For example, the runtime of the loops can deviate by some milliseconds. Added up for all loops, this results in some seconds postponement of the workload in every minute. This means that the workload does change considering a one minute time interval.

For a higher intensity of workload per minute the resulting values for DBSE427 do not fit. They are much smaller than expected. In these regions the execution time of the statements, which contain the sorts has an increasing impact on the overall runtime. For the basic workload, the overall statement execution time is about 10 seconds. For 400% workload it is 42 seconds and for 800% 84 seconds. The high intensity workloads execute much more statements per minute. The statement execution time, which also means the sort time, has a much higher impact on the overall runtime than for the low intensity workloads.

This means that for these workloads fewer statements are performed per minute as supposed to, which results in fewer sorts as predicted. This fact has to be considered when adjusting the waiting time. In the waiting time the longer overall execution time has to flow in.

To correct these values for higher amount of workload, reruns for the tests are done. The additional runtime is split to every loop and subtracted from the waiting time. For the workload with a lower number of performed statements it is not necessary to redo the tests because the overall impact of the statement execution time is small.

The new results are more exact but for 400% and 800% the values differ for more than 8 of the expectation. The main reason is the execution time of the sorts. This time changes in every test run. The 800% workload performs 5600 loops and in this case even a few milliseconds difference per loop have a high impact on the workload per minute. This non-deterministic behavior makes it impossible to compute the correct waiting time and this results in values which deviate from the expected ones.

These deliberations lead to the conclusion that there is an upper bound for the number of sorts. For example, if four sorts need 30ms time to be finished, it is not possible to process more than 8000 sorts (2000 loops) per minute. This bound does depend on the system and on the sorts but there has always to be an upper bound.

The expected behavior of DBSE427 is linear until the upper bound is reached. Increasing the number of sorts in the workload scenario should increase the number of sorts computed by OPM for the same amount. As soon as the upper bound is reached the number of sorts should remain constant and changes in the scenario which aim to execute more sorts per minute do not have effects.

In Figure 6.1.3 the results of the test runs up to 800% workload are pictured in a diagram. The x-axis is the workload in percent and the y-axis is the number of sorts per minute. The uncorrected values are black, the corrected values are green, and the expected values are red.

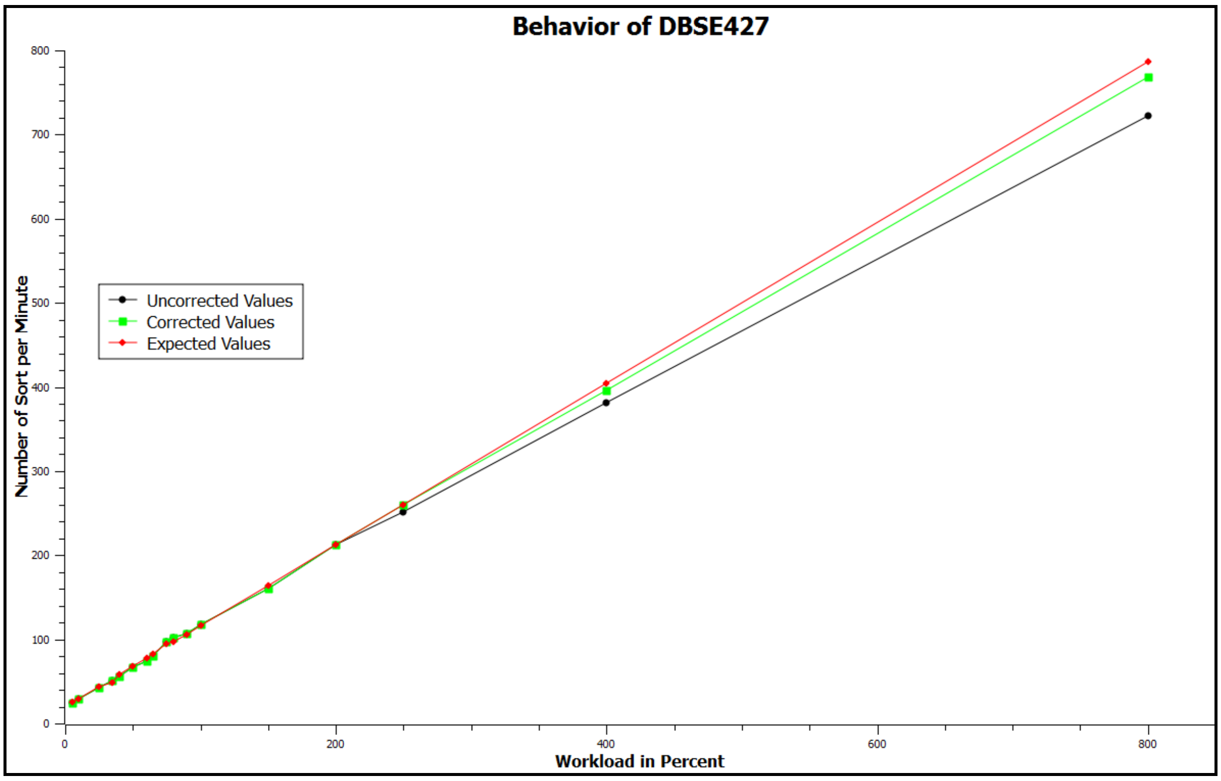


Figure 6.1.3: Uncorrected, Corrected, and Expected Values for DBSE427

For the low intensity workloads the values of the tests are very close to the expected values. In the higher regions there is a gap between the expected and real values. As mentioned above, this can be explained by the changing sort runtime. To close this gap it would be necessary to know the exact execution time of a sort before it is performed and adjust the waiting time according to this. This is not practicable.

It can be seen that the results meet the expectation. There are inaccuracies for tests with 75% and 80% workload. These values are slightly higher than expected. Reruns of these tests showed that the values pictured in the diagram are outliers probably caused by environmental impact.

During tests to find the upper bound for the number of sorts per minute it became apparent that the runtime is also depending on the DB2 instance. The used instances are located on the same server and are the same DB2 version. The execution time for one loop differs between the instances partly for more than 25ms. Taking this into account the tests are done to prove that there is an upper bound but this upper bound is not exactly determinable.

Workload is executed with 35000 loops and 40000 loops and zero milliseconds waiting time. On the used instance one loop lasts about 30ms. Each minute 2000 loops which means 8000 sorts should be performed. For 35000 loops OPM monitors about 7600 sorts. This value can be explained by looking at the loop runtime which differs slightly. This runtime can be seen in the log files of the workload tool. Some loops need 31ms or 32ms, which leads to the awareness that even for one instance the upper bound is not exactly determinable. Figure 6.1.4 shows the extended linear function for the number of sorts. The two green data points are the actual values for 35000 loops and 40000 loops.

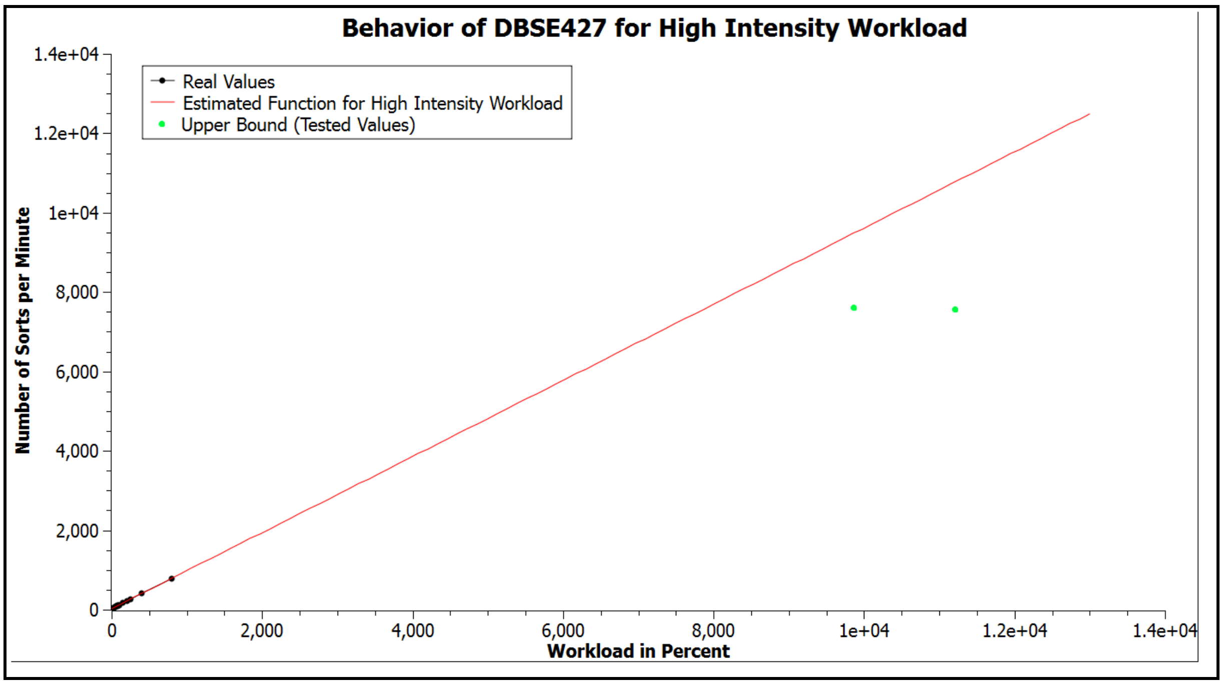


Figure 6.1.4: Upper Bound for DBSE427

The result for 40000 loops is the same as for 35000 loops. OPM monitors again around 7600 sorts per minute. The overall runtime is longer for this workload but the values of the metric remain the same.

The overall result of the semantic validation of the metric DBSE427 is that this metric is considered to be implemented correctly in the tested OPM version. The expected linear behavior can be seen in the diagram and the predicted upper bound exists. Still, the actual values deviate from the expected values. This can be seen in Table 6.1.1. The large majority of the deviations are within 0% and 5%. Only one value exceeds the 5%.

This means, that although the metric DBSE427 is considered to be implemented correctly the real values deviate from the expected values. Transferred to future validation of new OPM versions, this results in the insight that for correct implementations of DBSE427 there is an allowed deviation of 5%. When validating DBSE427, this deviation has to be satisfied by the majority of the computed values to draw the assumption that the metric is implemented correctly.

Workload in Percent	Expected Value for DBSE427	Deviation of the Expected Value in %
800	786.68	2.31
400	404.00	1.97
250	260.00	0.25
200	212.00	0.00
150	164.06	2.27
100	116.00	1.43
90	105.71	1.18
80	97.42	4.34
75	95.00	1.65
65	82.34	3.30
60	77.14	3.85
50	68.00	2.65
40	58.40	3.66
35	48.92	5.29
25	44.00	3.61
10	29.60	1.32
5	24.90	4.98

Table 6.1.1: Expected Values and Deviation of DBSE427

In the very low and very high regions of the workload it can happen that this 5% is exceeded although the metric is correct. This is caused by the different impact of the runtime of the sorts. Exceeding of the 5% does not always mean that a defect occurred. Especially, for low exceeding it has to be checked if the workload causes this deviation.

This deviation is used as a reference point to validate DBSE427 in future testing. Depending on the system on which the tests are performed this deviation might have to be adjusted. For example, if the difference between the runtime of sorts is very large.

In the next section the metric sort overflows is validated. Correctness criteria are derived and tests to check whether they are fulfilled are performed.

6.1.3 Semantic Validation of Sort Overflows

The ratio of sort overflows depends on several parameters. It is influenced by the number of total sorts, by the number of sorts causing overflows and this number depends on the size of the sort heap.

The correct implementation of DBSE427 is the basis for the semantic validation of sort overflows. Further, different preferences of all parameters are tested and the results are used to describe the correct behavior.

The same test runs for validating DBSE427 are used for the first test of sort overflows. Every sort of the workload scenario is causing an overflow. There is enough memory available for the sorts of OPM. Otherwise the percent value of sort overflows would be 100.

The expected behavior for this test is a nearly linear growth for the lower amount of workload. If a workload of low amount is increased the enhancement of the total number of sorts is big compared to the overall number of sorts. Enhancing a higher amount of workload does not have the same impact to the overall sorts. This expectation can be explained with the following example. The total number of sorts of the 10% workload are increased by about 48% for the 25% workload. For the 10% workload 29.60 sorts are expected, for the 25% workload 44.00 sorts are expected. This is an increase of 14.4 sorts, which are 48% of 29.60 sorts. Changing the 75% workload to the 90% workload is an enhancement of the total number of sorts by about 12%.

This fact leads to the assumption that increasing the workload more and more the values should follow an asymptotic function. They should approximate 100% but never reach it because there are sorts of OPM which do not cause overflows. In Figure 6.1.5 the black line pictures the result values. The red line is the expected asymptotic function.

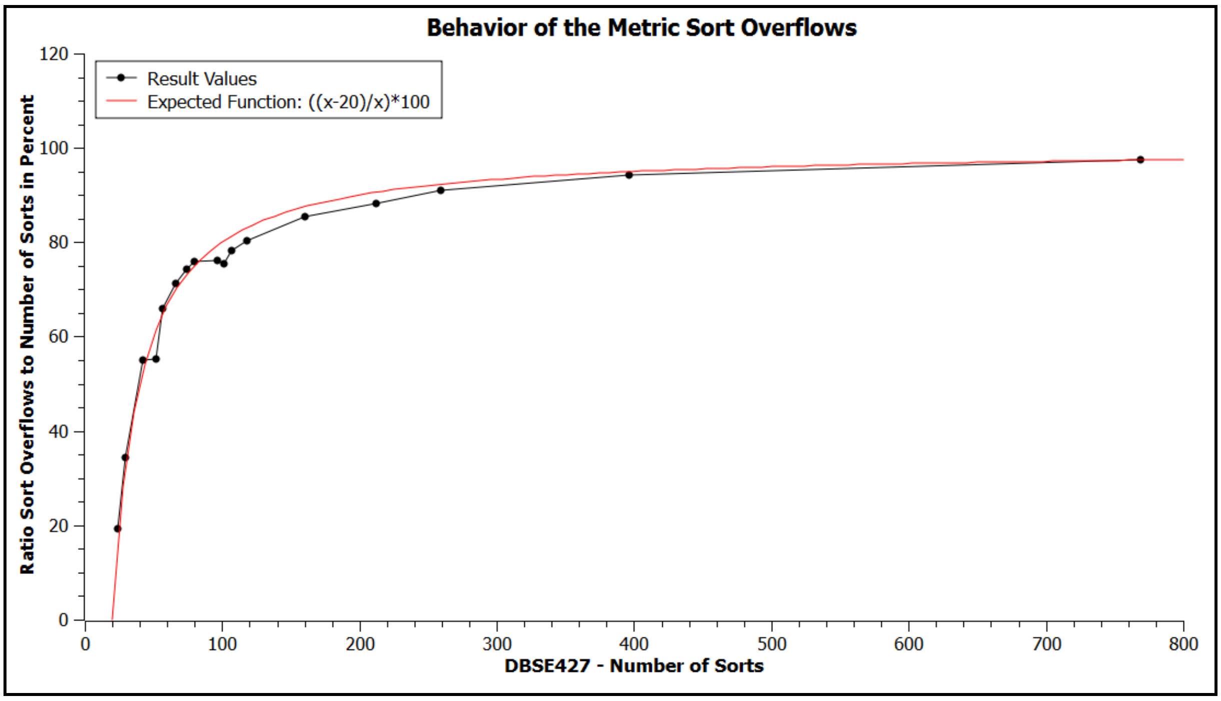


Figure 6.1.5: Results of Sort Overflows

The graph reflects the expected behavior. For low numbers of sorts there is a nearly linear growth and for increasing numbers of sorts the function approaches 100. The lowest number of possible sorts is 20. These are performed by OPM itself. For 20 sorts there would be no overflows because the size of the sort heap is sufficient. The expected function can be computed exactly.

The metric sort overflows is the ratio of sorts causing overflows to the total number of sorts. In this case the total number of sorts is x and the number of sorts causing overflows is x-20 because the 20 sorts executed by OPM do not cause overflows. The ratio is ((x - 20) / x) and because the value is in percent the results have to be multiplied by 100. This leads to the exact function f(x) = ((x – 20) / x) * 100.

This correctness criterion is globally valid and not depending on a specific workload. If different workloads and different environments are used, it has to be checked if OPM still executes 20 sorts. There is the possibility that this value changes if the data OPM is monitoring increases. In this case, the expected function has to be adapted. This does not influence the correctness criterion in general. Increasing the sorts causing overflows always results in an asymptotic behavior if the number of sorts which do not cause overflows remains the same.

The next criterion which is proved is the behavior of sort overflows when increasing the number of sorts which do not cause overflows. The workload scenario for the basic workload of the earlier tests is modified. In the preparation phase of the scenario a new table is created including two rows and a new SQL statement is added to the main loop. It performs one simple sort, which does not cause an overflow. This statement is surrounded by a parameterized loop to be able to change the count of executions.

The expected values follow a function which is the opposite of the one pictured in Figure 6.1.5. It starts with a high ratio of overflows and approaches to the x-axis for incremented numbers of sorts. For a low number of total sorts the proportion of big sorts causing overflows is high. During enhancement of the number of sorts, which do not need too much memory this proportion becomes less. The values will never reach the x-axis because there are always sorts causing overflows. In Figure 6.1.6, the graph of the result values is shown. The red line is the expected asymptotic function.

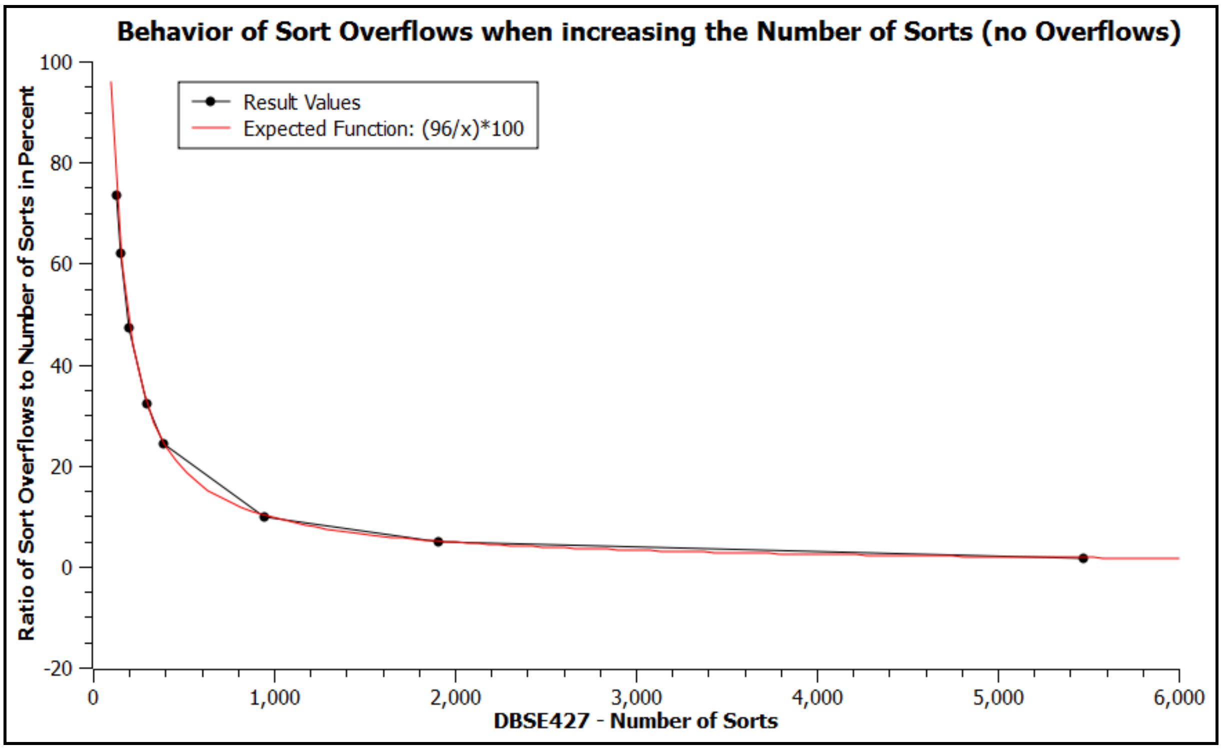


Figure 6.1.6: Sort Overflows when increasing Sorts with no Overflows

The computed values follow the expected behavior. The asymptote is 0 which will never be reached because the used workload scenario performs sorts which cause overflows. For low values of DBSE427 the ratio of overflows declines nearly linear.

The slope is depending on the sorts which need more memory than the sort heap offers. These sorts are needed otherwise there would be no sort overflows and the metric returns 0 for all values of DBSE427. This means that this correctness criterion can only be used if there are sorts causing overflows. In this case, the result values always follow an asymptotic function, which slope is depending on the workload scenario.

For every workload scenario, which is used for this test, the expected function can be computed exactly. The argumentation is the same as for the function computed in the previous section. The difference is that the number of sorts causing overflows is constant and the number of sorts which do not cause overflows is changed. This lead to the function f(x) = (a/x)*100 where a is the number of sorts causing overflows. In Figure 6.1.5 a equals 96 because the basic workload scenario is used.

For both asymptotic behaviors the allowed deviation of the expected value for future validation is set to 7%. The argumentation is the same as for the 5% deviation of DBSE427. The metric sort overflows is considered to be implemented correctly because of the results of previous tests. The expected values and the deviation of the actual values of the deviation are computed and shown in Table 6.1.2.

The large majority of the values deviates between 0% and 7%. There are only three values which exceed the range of 7%. For future validation this means, that if the metric sort overflows is correctly implemented, the majority of the values is deviating up to 7% of the expected value. Otherwise the assumption has to be drawn that the metric is incorrect.

Workload in Percent	Expected Values of Sort Overflows	Deviation of the Expected Value in %
800	97.40	0.00
400	94.95	0.73
250	92.29	1.31
200	90.56	2.69
150	87.53	2.47
100	83.00	3.23
90	81.29	3.65
80	80.32	6.16
75	79.29	4.06
65	74.88	1.44
60	73.03	1.59
50	69.79	2.03
40	64.45	2.36
35	61.17	9.73
25	52.84	4.09
10	31.53	9.17
5	16.60	16.71

Table 6.1.2: Expected Values and Deviation of Sort Overflows

The third correctness criterion for sort overflows deals with the sort heap size. The size of the sort heap declares the size of the memory useable for each sort. If this size is too small an overflow is caused. The expected value distribution is illustrated in Figure 6.1.7.

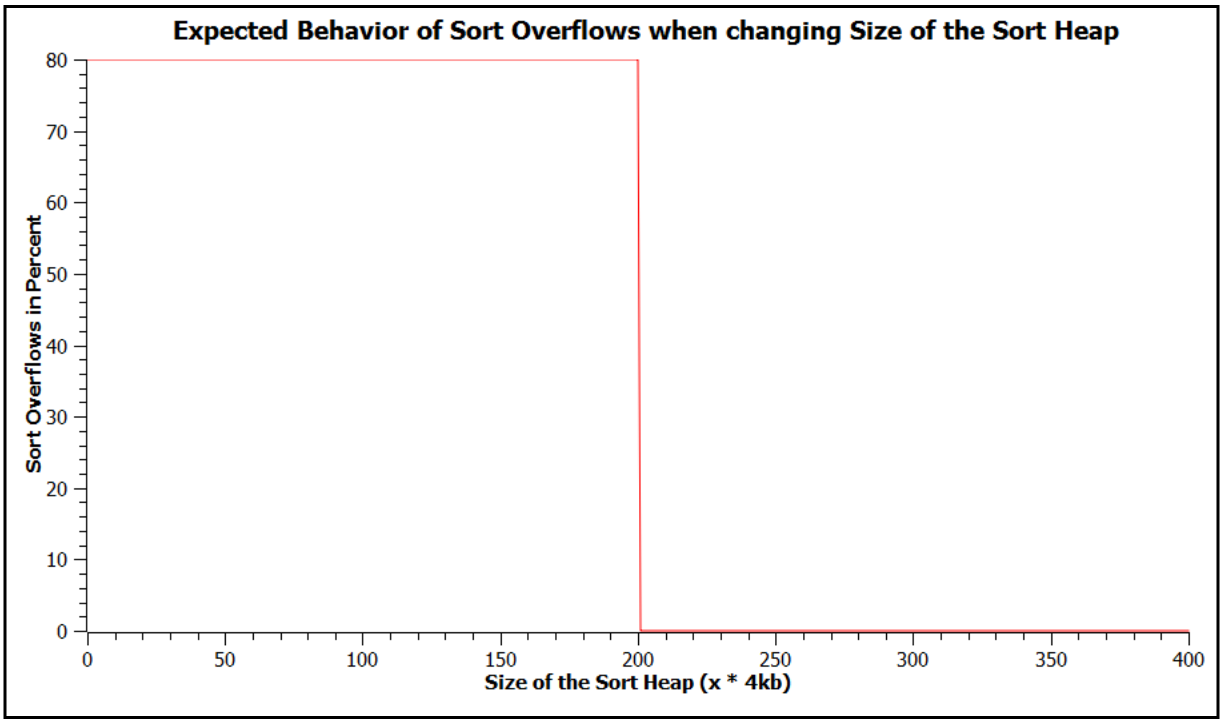


Figure 6.1.7: Expected Behavior of Sort Overflows Changing Sort Heap Size

The expected behavior is a constant sort overflow value until a certain size of the sort heap is reached (in this designed example 200). Before this size the memory is too small for the executed sorts. 100% overflows are not reached because of the sorts executed by OPM itself. As soon as the sort heap exceeds this size it is always big enough and the sorts do not cause overflows. The percent of overflows is expected to drop to 0.

The real behavior of the sort overflow values are pictured in the graph of Figure 6.1.8. It can be seen that the values follow the expected distribution for most sizes of the sort heap. In the part marked red there are some values which behave not as predicted. The overflows in this part increase to about 20% although the sort heap size is big enough to provide sufficient memory.

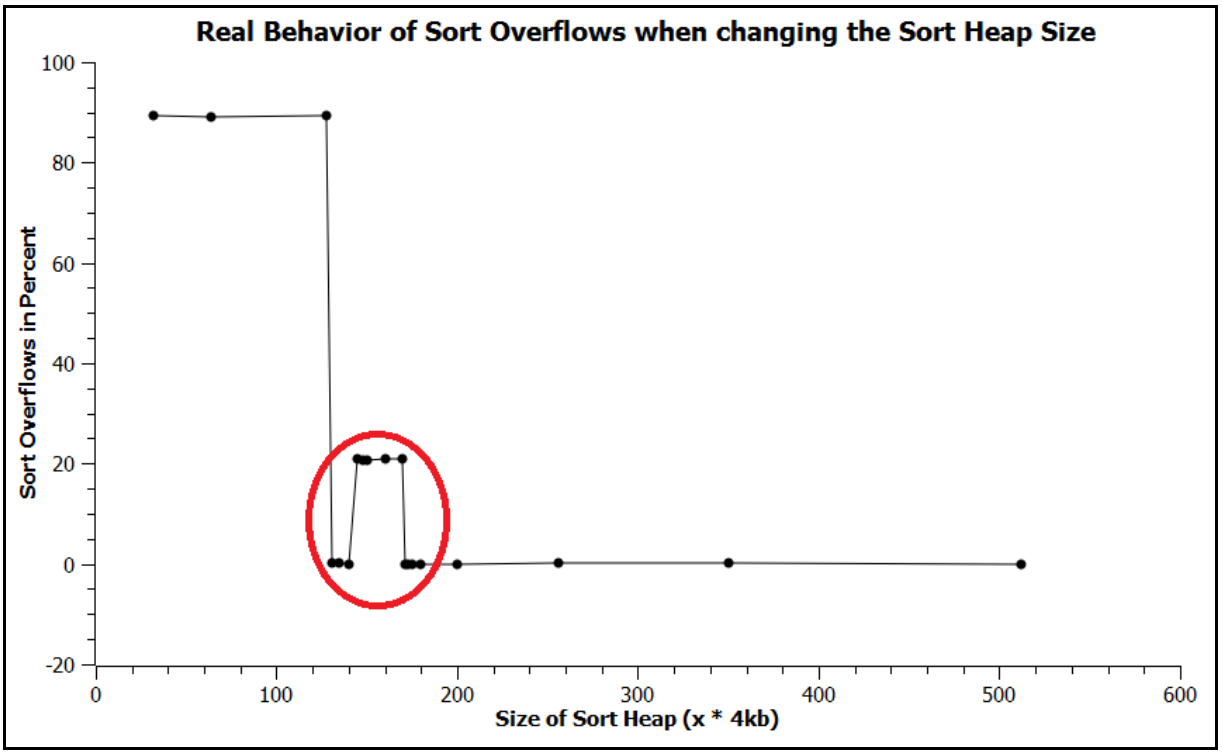


Figure 6.1.8: Real Behavior of Sort Overflows changing Sort Heap Size

At the moment there is no real explanation for this behavior. It is most probably caused by DB2 because for this size of the sort heap the workload does not execute sorts which cause overflows. The sorts of OPM do not cause overflows either. To be able to use this validation test the behavior shown in Figure 6.2.8 has to be explainable.

For this correctness criterion there two allowed deviations of the result values of the expected values. The first is set to 7% for the sizes of the sort heap which cause overflows. In this case, the values of the metric behave in the same way as for the correctness criteria before. The second deviation is 0% for the sizes of the sort heap when no overflows are caused. In this value range the behavior of the metric sort overflows is deterministic. This means that a behavior as seen in Figure 6.1.7 needs to be investigated. In Table 6.1.3 the correctness criteria for the metrics DBSE427 and sort overflows when using statistical metamorphic testing are listed.

Metric	Workload	SMT Criteria
DBSE427	1. Changing number of sorts	1. Linear growth
	2. Changing number of sorts	2. Upper bound
Sort Overflows	1. Increasing number of overflows	1. Asymptotic growth (asymptote = 100), f(x) = ((x-20)/x)*100
	2. Increasing number of sorts (no overflows)	2. Asymptotic growth (asymptote = 0), f(x) = (a/x)*100
	3. Increasing size of sort heap	3. Constant overflows until sort heap size big enough, then constant 0

Table 6.1.3: Summary of the Correctness Criteria

The allowed range for the metrics is determined based on the assumption that the computed values are correct. Although, these values are considered to be correct, there is a deviation of 5% respectively 7% of the expected values. This leads to the conclusion that the values of the metrics in future validation have to be within these ranges, to consider the metrics as correct.

Within the allowed range there are possible wrong values. If the metric is considered to be correct there is no guarantee that it is implemented correctly. As discussed in Chapter 3 and Section 5.2.2 testing in case of software cannot ensure the absence of defects. It is not able to validate every possible state of the software. The coverage is limited by the used test cases. However, if a test failed, the probability that a defect occurred is very high.

According to the definition of statistical metamorphic testing (see Section 5.2.3), a metric is considered to be correct if it passes the majority of the statistical hypothesis tests. There is the possibility that it passed the majority of the tests, but for the failed tests the values have a very high deviation of the expected values. In this case, the metric passed the validation but further action may be required. The framework has to indicate the user to this behavior.

The next section describes the implementation of the prototype. The available test environment provides the functions for accessing and configuring OPM and storing the test results.

6.2 Prototype

The prototype of the Semantic Validation Framework is integrated into the test environment. For the prototype a new test scenario is built. A test scenario is represented by an XML file containing all the tasks which are performed. To start the semantic validation this scenario has to be started. Figure 6.2.1 shows the procedure of the prototype.

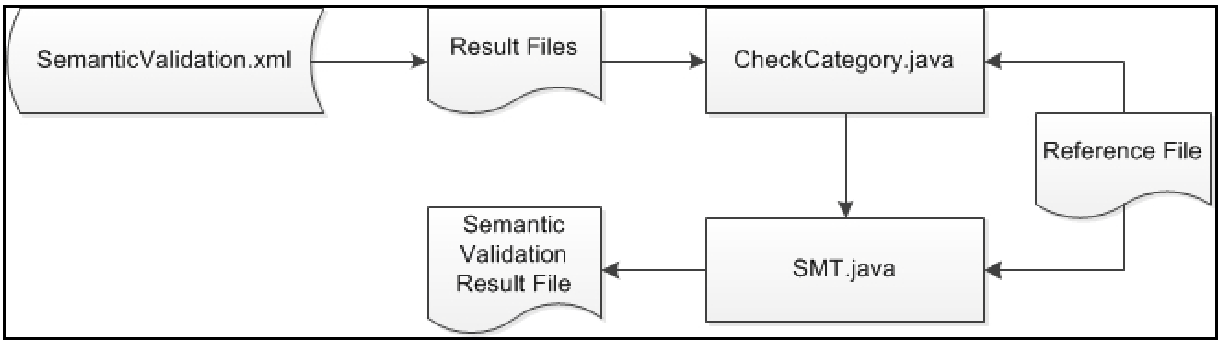
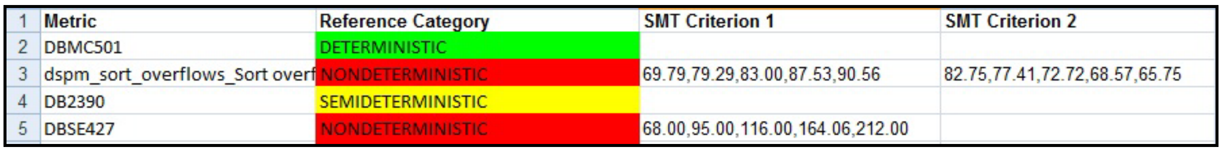


Figure 6.2.1: Validation Process with the Prototype

After the test scenario *SemanticValidation.xml* is started, it executes all the necessary tasks. These tasks include a preparation phase where the database is added and the monitoring profile is configured. The workload is started as well as data retrieval after the workload is finished. Additionally, the tasks contain the handling of the result files and the semantic validation.

After the workload is finished and the data is stored in the result files, *CheckCategory.java* is executed. This java class is responsible to classify the actual values and compare the computed category with the reference category. The reference category is stored in the Reference File. This is an Excel file including all metrics, the reference class from the earlier classification, and correctness criteria for statistical metamorphic testing. Since, the result files of the test environment are Excel files, the functions to handle these files can be used for the reference file as well. Further, changes in the reference file, for example, adding new correctness criteria, can be done with little effort. In Figure 6.2.2 an extract of this file can be seen.



1	Metric	Reference Category	SMT Criterion 1	SMT Criterion 2
2	DBMC501	DETERMINISTIC		
3	dspm_sort_overflows_Sort overf	NONDETERMINISTIC	69.79,79.29,83.00,87.53,90.56	82.75,77.41,72.72,68.57,65.75
4	DB2390	SEMIDETERMINISTIC		
5	DBSE427	NONDETERMINISTIC	68.00,95.00,116.00,164.06,212.00	

Figure 6.2.2: Extract of the Reference File

For sort overflows in row 3 and DBSE427 in row 5 correctness criteria for statistical metamorphic testing are available. Criterion 1 for sort overflows is the behavior when the number of sorts causing overflows is increased. Criterion 2 are the values when the number of sorts causing no overflows is increased. The criterion of the metric when changing the size of the sort heap size is not saved in the Reference File at the moment because there is no explanation for the behavior (see Section 6.1.3). The only criterion in the file for DBSE427 are the values when changing the number of sorts. The upper bound is checked automatically within the prototype.

The different values of the correctness criteria are separated by comma. For criterion 1 these values are the expected output for 50%, 75%, 100%, 150%, and 200% workload, which have been computed in Sections 6.1.2 and 6.1.3. Criterion 2 are the expected values for 0, 8, 16, 24, and 32 sorts, which are causing no overflows executed together with the basic workload. The process of validation using criterion 2 is explained in Section 6.1.3. The number of values is not limited to 5 values per criterion. If more workload is executed, the expected values can be inserted in the corresponding cells at the right place. The values are sorted from low intense to high intense workload. The actual workload for the prototype is chosen to cover low and high intensity.

Since the Reference File is used for all metrics to check the category, there are two more metrics in the figure. Both are not non-deterministic and they have no statistical metamorphic criteria. At the moment the prototype is only used to validate the metrics sort overflows and DBSE427.

If the metric passed the category check, the class *SMT.java* is called and the statistical hypothesis tests are performed. The criteria are read from the Reference File and split into the single values. These values are tested against the actual values received from OPM. If the majority of the actual values does not deviate for more than the allowed range (5% for DBSE427 and 7% for sort overflows) of the reference values, the metric passed the test (see Section 5.2.3). If it passes all tests, it is considered to be implemented correctly.

For every tested metric an entry in the Semantic Validation Result File is created. It includes the overall result whether the metric is semantically correct. Further, it contains the results of the category check and of each statistical hypothesis tests. If a metric did not pass, the user can see which tests failed. Additionally the largest deviation of the expected value is stored and there is information about how many tests failed.

In Chapter 7 the prototype is tested with a new version of OPM and the results of the thesis are discussed.

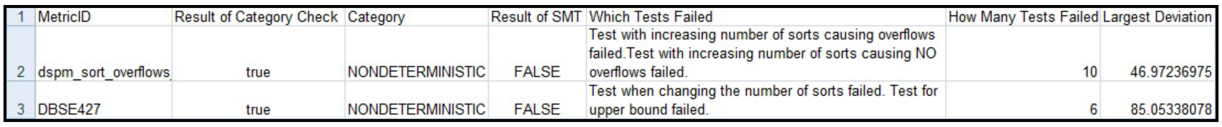
7 Evaluation

The first part of Chapter 7 consists of the evaluation of the prototype, which is introduced in Section 6.2. The prototype is used to validate the metrics DBSE427 and sort overflows for the latest version of OPM, which has been recently shipped to the customer. The results are investigated and necessary adaptations are presented. The second part is a discussion of the overall results of the thesis. The benefits and shortcomings of the chosen approach are pointed out and it is shown how the framework changes the testing process of OPM.

7.1 Evaluation of the Prototype

The prototype is tested on a new version of OPM. This new version is installed on the same server as the version used for deriving the correctness criteria. The version of the underlying DB2 system is the same but the version of the fix pack and the used instance differ. This setup is chosen to investigate how the prototype of the framework is behaving in a different environment. The test is used to check if the assumptions made for the framework are valid when the preconditions (OPM version, DB2 instance, DB2 fix pack version) change.

The prototype executes the scenario *semantic_sort_ms* with 12 different parameter combinations. Each combination is run for 3 times to receive 30 values. According to the "rule of thumbs" mentioned earlier in Section 4.2.3, 30 values are enough to reduce the influence of outliers. This ends in 36 workload executions. One complete run of the prototype takes about 18 hours. With the created data it is possible to validate the correctness criteria for DBSE427 and sort overflows. Figure 7.1.1 shows an extract of the result file for the semantic validation of the first test run.



1	MetricID	Result of Category Check	Category	Result of SMT	Which Tests Failed	How Many Tests Failed	Largest Deviation
2	dspm_sort_overflows	true	NONDETERMINISTIC	FALSE	Test with increasing number of sorts causing overflows failed.Test with increasing number of sorts causing NO overflows failed.	10	46.97236975
3	DBSE427	true	NONDETERMINISTIC	FALSE	Test when changing the number of sorts failed. Test for upper bound failed.	6	85.05338078

Figure 7.1.1: Results of the First Test Run

It can be seen that both metrics pass the category check but fail in each statistical hypothesis test. The maximum deviations of the expected values are about 47% for sort overflows and about 85% for DBSE427. This very high deviation indicates a defect in OPM, errors in the prototype, or wrong assumptions.

For further investigations the data received from OPM is checked. The values in the result files do not fit to the expected values in the Reference File. All of the values, which do not fit are too small. This means that the overall result of the prototype, the metrics did not pass the tests, is correct for now. The values computed by OPM do not fulfill the correctness criteria developed in Chapter 6. Either there are defects in OPM or wrong assumptions have been made.

To find out what causes the actual values to have this large deviation, further test runs are performed. These test runs are done to figure out how the system and the OPM version behave in the new environment. In this case, two different runs are accomplished.

The first one executes the workload scenario *just_wait_ms* to find out what activity is caused by the new OPM. The result of this run is that instead of around 20 sorts per minute, OPM is performing only 16 sorts per minute.

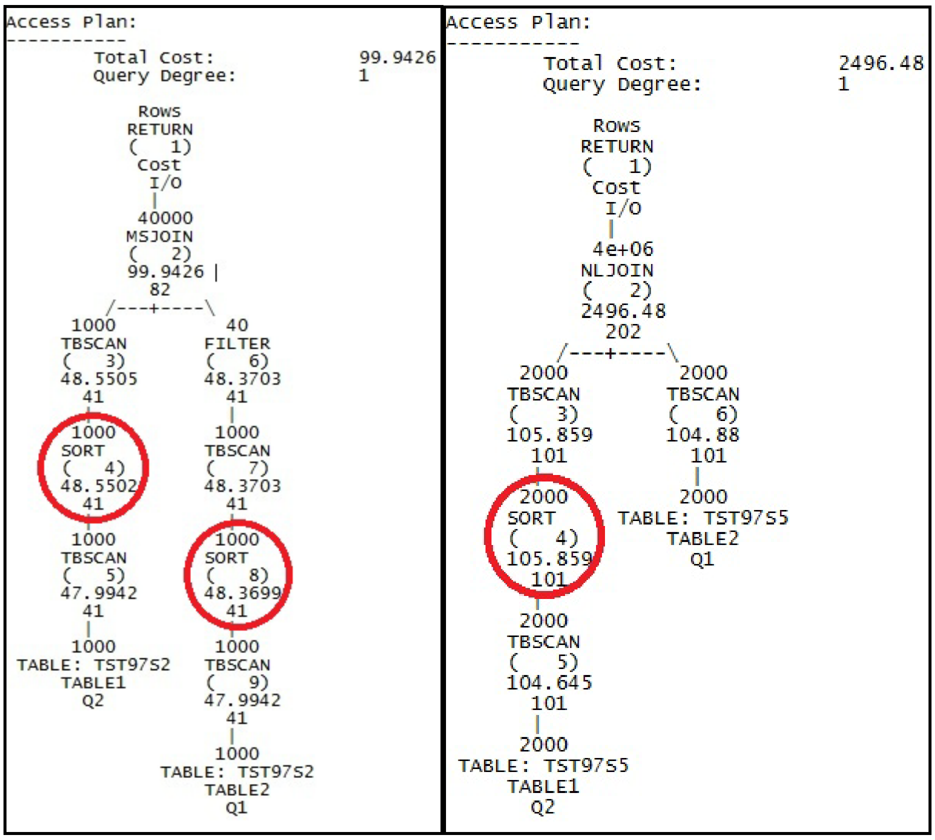
The second run executes the basic workload. This is done to see how many sorts per minute the system is doing for this workload. The result of this run serves as a basis to compute the expected values for the other workloads. As discussed in Chapter 6 the runtime of one loop of the scenario *semantic_sort_ms* differs between different DB2 instances. With the insights of the additional test runs these differences can be considered for the expected values.

To find out the number of sorts per minute, the log file of the workload tool is checked. The log reveals 24 loops per minute. This means that around 112 sorts per minute should be monitored by OPM (24 loops multiplied by 4 sorts plus 16 sorts of OPM). The actual values are most of the time the half of the expected values. There are few which are exactly as expected.

The sorts executed by the workload scenario are the same for every one minute time interval. This leads to the conclusion that the two sorts done by DB2 during the inner join are not performed in each loop. To prove this the two statements containing the inner join (see Figure 6.1.2) are replaced with four new statements, which select two values from a table without joining tables and sort these values. For these statements DB2 does not perform any additional sorts.

The result of this test is a stable distribution of the number of sorts per minute around the expected value of 112 sorts. It seems that the newer fix pack of the database system DB2 leads to different decisions about the execution process of inner joins.

To confirm this assumption, the access plans for both instances of the SELECT statements, which are executing the inner joins, are examined. On the DB2 instance with the older fix pack version, the optimizer is using a merge scan join, which requires sorted input. This results in two sorts per statement, as described in Section 6.1.1. On the instance with the newer fix pack version, the optimizer is choosing a nested loop join. This join does not require sorted input but there is the possibility that the data is sorted. This decision is done by the optimizer [32]. In Figure 7.1.2 the access plans are shown.



```
Access Plan:
-----------
        Total Cost:             99.9426
        Query Degree:           1

             Rows
            RETURN
            (   1)
             Cost
              I/O
              |
             40000
            MSJOIN
            (   2)
            99.9426 |
              82
          /---+----\
      1000           40
     TBSCAN        FILTER
     (   3)        (   6)
     48.5505       48.3703
       41            41
       |             |
      1000          1000
     SORT          TBSCAN
     (   4)        (   7)
     48.5502       48.3703
       41            41
       |             |
      1000          1000
     TBSCAN        SORT
     (   5)        (   8)
     47.9942       48.3699
       41            41
       |             |
      1000          1000
 TABLE: TST97S2    TBSCAN
     TABLE1        (   9)
       Q2          47.9942
                     41
                     |
                    1000
             TABLE: TST97S2
                   TABLE2
                     Q1
```

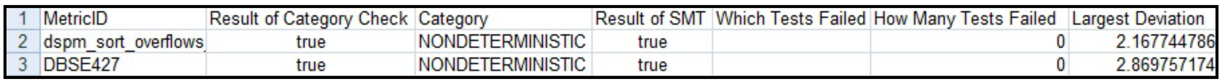
```
Access Plan:
-----------
        Total Cost:             2496.48
        Query Degree:           1

             Rows
            RETURN
            (   1)
             Cost
              I/O
              |
             4e+06
            NLJOIN
            (   2)
            2496.48
              202
          /---+----\
      2000           2000
     TBSCAN         TBSCAN
     (   3)         (   6)
     105.859        104.88
       101            101
       |              |
      2000           2000
     SORT       TABLE: TST97S5
     (   4)         TABLE2
     105.859          Q1
       101
       |
      2000
     TBSCAN
     (   5)
     104.645
       101
       |
      2000
 TABLE: TST97S5
     TABLE1
       Q2
```

Figure 7.1.2: Access Plans for the SELECT Statements

The sort operations are marked red. The access plan for the old version is on the left and contains two sorts. On the right is the access plan for the new version. In this case, the nested loop join has been chosen and no additional sorting has been done. This results in only one sort operation.

The workload scenario has to be adapted to this new behavior. For further testing the scenario with the two replaced statements containing the inner joins is used to be independent on the database system. The correctness criteria are not influenced by this decision. Nevertheless, the changed number of sorts done by OPM itself has to be considered and the values in the reference files have to be adjusted.

The prototype is tested again using the adapted workload scenario and the updated Reference File. Both metrics passed all tests. This means, they are considered to be implemented correctly in the new version of OPM. In Figure 7.1.3 an extract of the semantic validation result file of this test is shown.



1	MetricID	Result of Category Check	Category	Result of SMT	Which Tests Failed	How Many Tests Failed	Largest Deviation
2	dspm_sort_overflows	true	NONDETERMINISTIC	true		0	2.167744786
3	DBSE427	true	NONDETERMINISTIC	true		0	2.869757174

Figure 7.1.3: Results of the Adapted Test Run

The evaluation reveals that it is necessary to perform the additional test runs to calibrate the framework whenever a new OPM version is tested or a different environment is used. If these *calibration runs* are not done there is a high probability that the framework computes wrong results. The updated validation process with the prototype including the calibration can be seen in Figure 7.1.4.

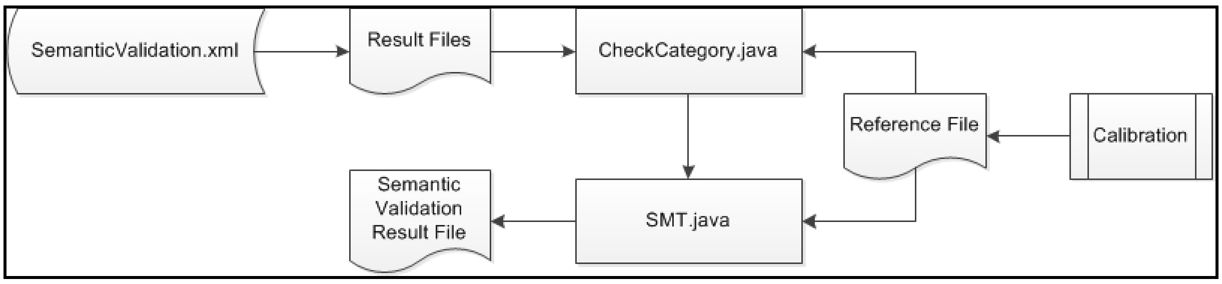


Figure 7.1.4: Validation Process of the Prototype Including Calibration

The calibration includes the two test runs mentioned above. One run to determine the action performed by OPM and the other run to find out how the workload behaves on the monitored DB2 instance. Based on the result of these test runs, the expected values are computed using the correctness criteria developed in Chapter 6 and saved in the Reference File

For example, if OPM is performing 10 sorts per minute and the test with the basic workload results in 100 sorts per minute, the correctness criterion for DBSE427 can be computed as follows: The expected value for the 50% workload is 100 sorts per minute divided by 2, because the workload is only half as intense as the basic workload, plus 10 sorts per minute done by OPM. The result of this computation is 60 sorts per minute. Further values can be determined accordingly. At the moment the calibration is done manually and takes about one hour.

The next section contains a discussion about the results of the thesis. It is demonstrated which goals have been achieved and where problems are present.

7.2 Discussion

The goal of this thesis is to develop a framework to automatically perform a semantic validation of the OPM performance metrics. The framework needs to be robust against changes in OPM. This means updates in the code should not affect the semantic validation process. No user interaction during the tests and a simple way to add and remove metrics for semantic validation are desired properties. Additionally, the tests have to be fast to be able to execute them on a regular basis, the results have to be meaningful and simple interpretable and it has to be possible to execute the semantic validation without much foreknowledge.

The developed framework is able to validate the OPM performance metrics semantically. The result of the prototype evaluation shows that it is possible to receive assertions about the semantic correctness without user interaction during the validation. Manual steps are only necessary for the calibration runs. To start the semantic validation the according test environment scenario is launched. This can be done by inexperienced users. After the test run is finished the user checks the result file. This file offers assertions about the correctness for each tested metric. There are information about the number of failed tests and the largest deviation of the expected value. If a metric did not pass, a description is provided, telling the user which test failed.

The framework handles the whole OPM environment as a black box. Changes in the code of OPM do not influence the semantic validation in the large majority of the cases. Updates of OPM, altering the activity, which OPM is causing on the monitored instance, may lead to necessary adaptations of the reference values. As described in the section of the prototype evaluation, the number of sorts performed by OPM decreased to 16. In this case, the reference values have to be updated.

To notice such changes, calibration runs are necessary when validating a new version of OPM or testing in a different environment, including DB2 version, DB2 instance, server, etc.. The runtime of these calibration runs averages about one hour. If no calibration is done, it is more likely to receive wrong results which may lead to a large effort spent on diagnostics.

At the moment, a run of the prototype takes about 18 hours to be finished. It is possible to perform the validation on a regular basis. Though, the prototype validates only two metrics at the moment, the workload which is executed can be used for more metrics without raising the runtime. The workload scenario *semantic_sort_ms* creates data for more than 25 metrics of the workload dashboard. Additionally, it covers many metrics of other dashboards. To validate these metrics correctness criteria have to be derived and the metrics have to be added to semantic validation.

New metrics are added to semantic validation by creating a new entry in the Reference File. An entry consists of the name of the metric, the reference category, and correctness criteria. The reference categories are available for all metrics of the workload dashboard. For every other dashboard a classification of the metrics has to be done.

To derive the correctness criteria profound knowledge of DB2 and the semantic of the metric is needed. The behavior of the metric for different workloads has to be predicted and this behavior has to be validated. This process requires experienced users but once an entry in the Reference File for a new metric exists, the validation can be done with little foreknowledge.

The framework is able to make assertions about the correctness of metrics. This is done with the method statistical metamorphic testing. In common, testing can indicate the user to defects. It is not able to prove the absence of them. If the framework detects wrong values the user has to do further investigations. The framework gives answers which metrics did not pass and which of the tests failed.

Using static methods, introduced in Chapter 3, would lead to more detailed information about the detected defects than statistical metamorphic testing. Static methods make assertions for functions or for a model of the system and prove them to validate it. For this correctness proving no test run of the system has to be done.

Contrary to testing, static methods are able to check every possible path through the system under test. To implement these methods detailed knowledge about the code is essential. For the static method model checking, a model of the system has to be built, which requires expert knowledge about the system. Changes in the code directly influence static methods. It is necessary to check whether their implementation is still valid, every time changes have been done.

Another approach introduced in Chapter 3, which would provide more information about a found defect, is runtime assertion checking. For runtime assertion checking, assertions are implemented directly in the code and validated during runtime. This means that if a defect is detected it can be said exactly where this defect occurred.

Its implementation needs a deep understanding of the code and how to derive assertions, which can prove the absence of errors. Another problem of directly implementing the assertions in the code appears when the code is modified. In this case, the assertions have to be modified and adapted to the new code. This cannot be done by inexperienced users.

Using statistical metamorphic testing in the framework offers the possibility to perform semantic validation with little adaption effort when testing new versions of OPM. It is able to detect the existence of defects. It provides fast, uncomplicated and profound testing based on validated correctness criteria. These criteria contain the semantic of the metrics, the input, and the output.

For deterministic metrics, which are DB2 parameters or values of the system, it is possible to decide exactly whether the metric is implemented correctly. Therefore, it is necessary that the user sets the parameter on the monitored DB2 instance to a concrete value. This value and the system values are stored in the Reference File. If the stored value is not received by the test environment there is a defect in the computation of the metric.

There are two shortcomings of statistical metamorphic testing. First, if a metric did not pass the tests it is necessary to do further investigations to find out what caused the metric to fail the tests. Second, if a metric passed all tests there is no certainty that there are no defects in the implementation of this metric.

For the second shortcoming the framework offers information to the user, which helps to realize that the behavior of the metric might be wrong. It computes the largest deviation of the expected values for every metric. If this deviation is very large although, the metric passed the semantic validation (the majority of the tests have been successful), it is recommended to investigate further. The size of the deviation, which may indicate a defect, cannot be determined in general. It depends on the metric and on the used environment.

Besides that, the framework is a significant improvement of the actual testing process of OPM. Without the framework, every test case to check if the received values are reasonable has to be done manually. It is impossible to achieve this for every metric and hence not done at the moment. Each workload has to be started and the right values have to be verified in the web interface. This process is time consuming and error-prone. The coverage of manual tests is very small compared to the possible number of metrics, which can be validated in one run of the framework.

The former automated testing was limited to simple range checking of percentage values and detecting exceptions invoked by the data access functions. The framework extends this former testing to validate the values computed by OPM for every metric. OPM has to be tested on different platforms. Using the framework all of these tests can be done simultaneously. There are scripts available, which automatically install and configure OPM on servers. It is possible to implement a function in these scripts to start the semantic validation. This would lead to very little user interaction for validating OPM metrics.

Overall, the framework improves the testing process for OPM. There is very few manual work needed to increase the number of test cases and tested metrics. It is possible to run additional tests, which have not been executed earlier due to lack of time.

The next chapter concludes this document. Further, it contains suggestions for possible future work.

8 Conclusion and Future Work

Validation is an important part in the field of software development. Validation is necessary to develop software, which is reliable and successful. It is time consuming and occupies many resources, but it saves costs by detecting defects before the software is shipped to the customer. To reduce the impact of validation to the total costs of software development a fast and automatic validation process is required.

The developed Semantic Validation Framework serves as tool for automatically validating the OPM performance metrics. It offers the possibility to validate these metrics without user interaction and provides understandable and meaningful results. The probability of finding defects is higher compared to the former validation process because the coverage of the metrics is much larger than with manual testing. More test cases can be done in the same amount of time. It is possible to run the framework on many servers at once.

To be able to use the validation framework in praxis, the prototype has to be extended. Functions to validate the deterministic metrics and semi-deterministic metrics are needed. The test environment provides the functionality to execute commands on servers. This allows to automatically set DB2 parameters for the deterministic metrics. If these parameters are set to a certain value, the framework is able to determine with certainty whether the corresponding metric is correctly implemented.

It is possible to automatically make calibration runs and insert the reference values in the Reference File. Therefore, one run of the workload and one run to find out the activity caused by OPM have to be executed and the values of the metrics have to be received. Based on these values, the remaining values for the correctness criteria can be computed and stored in the Reference File.

As mentioned in Chapter 7, there are scripts available to perform OPM installations and configurations automatically. The test environment offers commands to start its scenario from the command line. These commands can be integrated in the scripts.

This would mean that there is a way to fully automatically install, configure and validate OPM on several systems in parallel.

The most important part to be able to use the framework on a daily basis is the development of correctness criteria for further OPM metrics. The first step to derive these criteria is the classification of the metrics of the remaining dashboards. Corresponding workload is needed to classify the metrics and to serve as input for the validation. The derived correctness criteria have to be validated and saved in the Reference File.

Additional correctness criteria for DBSE427 and sort overflows can be derived to achieve more confidence about the correctness. In this thesis, only the intensity of the workload and one DB2 parameter are adjusted and used for validation. There are other possibilities. For example, changing the data in the used tables or combining different workload parameters.

Overall, the framework is a powerful tool to detect defects in OPM. It supports the test team in increasing the quality of OPM by covering much more metrics and test cases than it is possible with manual testing. The used method testing is able to validate all categories of metrics. For deterministic metrics an assertion about the correctness can be drawn with guarantee. In particular, statistical metamorphic testing is capable to handle non-deterministic behavior. This is important because the large part of the OPM metrics behaves non-deterministic and this behavioral pattern is very difficult to validate manually.

References

1. Papadakis, M., Malevris, N., et al.: Towards automating the generation of mutation tests. Proceedings of the 5th Workshop of Automation of Software Test (2010), p. 111-118
2. Chen, W.-J., et al.: IBM Optim Performance Manager for DB2 for Linux, UNIX, and Windows. International Business Machines Corporation (2011)
3. Osterweil, L., et al.: Strategic Directions on Software Quality. ACM Computing Surveys (CSUR) – Special ACM 50th-anniversary issue: strategic discussions in computing research (2006), p. 738-750
4. Lyons, J.: Linguistic Semantics: An introduction. Cambridge University Press (1995)
5. Fodor, J.: Semantics: an interview with Jerry Fodor, Revista Virtual de Estudos da Linguagem - ReVEL. Volume. 5, n. 8, 2007
6. Winskel, G.: The Formal Semantics of Programming Languages: An Introduction. The MIT Press, Cambridge, Massachusetts, London, England (2003)
7. Clarke, E.M., Wing, J.M., et al.: Formal Methods: State of the Art and Future Directions. ACM Computing Surveys (CSUR) – Special ACM 50th-anniversary issue: strategic directions on computing research. Volume 28 Issue 4 (1996), p. 626-643
8. Woodcock, J., Larsen, P.G., et al.: Formal Methods: Practice and Experience. ACM Computing Surveys (CSUR). Volume 41, Issue 4 (2009), Article 19
9. Fujita, M.: Model Checking: Its Basics and Reality. Design Automation Conference 1998. Proceeding of the ASP-DAC '98. Asia and South Pacific (2002), p. 217-222
10. Vardi, M.Y.: An automata-theoretic approach to linear temporal logic. Logics for Concurrency. Springer Berlin / Heidelberg (1996), p. 238-266
11. Mukherjee, A., Tari, Z., et al.: Memory Efficient State-Space Analysis in Software Model Checking. Proceedings of the Thirty-Third Australasian Conference on Computer Science – Volume 102 (2010), p. 23-32
12. Julliand, J., Masson, P.-A., et al.: Partitioned PLTL model-checking for refined transition systems. Information and Computation, Volume 207 Issue 6 (2009), p. 682-698
13. Model Checker SPIN. http://spinroot.com/spin/whatispin.html (2011)
14. Vardi, M.Y., Wolper, P.: An Automata-Theoretic Approach to Automatic Program Verification. Proc. 1st Symp. on Logic in Computer Science (1986), p. 332-344
15. Popovic, M., Kovacevic, V., et al.: A Formal Software Verification Concept Based on Automated Theorem Proving and Reverse Engineering. Engineering of Computer-Based Systems (2002), p. 59-66
16. Ouimet, M.: Formal Software Verification: Model Checking and Theorem Proving, Embedded Systems Laboratory Technical Report ESL-TIK-00214, Cambridge USA
17. Schumann, J.M.: Automated theorem proving in software engineering. Springer-Verlag Berlin, Heidelberg, New York (2001)
18. Parveen, T., Tilley, S., et al.: A Case Study in Test Management. ACM-SE 45 Proceedings of the 45th annual southeast regional conference (2007), p. 82-87

19. Bertolina, A.: Software Testing Research: Achievements, Challenges, Dreams. FOSE ’07 Future of Software Engineering (2007), p. 85-103
20. Ciupa, I., Leitner, A., et al.: Experimental Assessment of Random Testing for Object-Oriented Software. Proceedings of the 2007 international symposium on Software testing and analysis (2007), p. 84-94
21. Gerlich, Rainer, Gerlich, Ralf, et al.: Random Testing: From the Classical Approach to a Global View and Full Test Automation. Proceedings of the Second International Workshop on Random Testing (2007), p. 30-37
22. Ševčíková, H., Bleek, W.-G., et al.: Automated Testing of Stochastic Systems: A Statistically Grounded Approach. Proceedings of the 2006 international symposium on Software testing and analysis (2006), p. 215-224
23. Guderlei, R., Mayer, J., et al.: Testing randomized software by means of statistical hypothesis tests. Fourth international workshop on Software quality assurance: in conjunction with the 6th ESEC/FSE joint meeting (2007), p. 46-54
24. Clarke, L.A., Rosenblum, D.S.: A historical perspective on runtime assertion checking in software development. ACM SIGSOFT Software Engineering Notes (2006), p. 25-37
25. Chen, J., MacDonald, S.: Towards a better collaboration of static and dynamic analyses for testing concurrent programs. PADTAD ’08 Proceedings of the 6th workshop on Parallel and distributed systems: testing, analysis, and debugging (2008), Article No. 8
26. Giunchiglia, F., Shvaiko, P., et al.: S-Match: an algorithm and an implementation of semantic matching. Proceedings of ESWS (2004), p. 61-75
27. Mayer, J., Guderlei, R.: Test Oracles Using Statistical Methods. Proceedings of the First International Workshop on Software Quality, Lecture Notes in Informatics P-58, Köllen Druck+Verlag GmbH (2004), p. 179-189
28. Information / Explanation of metric maximum coordinating agents. http://publib.boulder.ibm.com/infocenter/tivihelp/v24r1/index.jsp?topic=%2Fcom.ibm.itcama.doc_6.2.3%2Fitcam_db2622126.htm (21.12.2011)
29. Information / Explanation of different DB2 agents. http://publib.boulder.ibm.com/infocenter/db2luw/v8/index.jsp?topic=/com.ibm.db2.udb.doc/admin/r0001208.htm (21.12.2011)
30. Zhou, Z.Q., Huang, D.H., et al.: Metamorphic Testing and Its Applications. Proceedings of the 8th International Symposium on Future Software Technology (ISFST 2004)
31. Guderlei, R., Mayer, J.: Statistical Metamorphic Testing – Testing Programs with Random Output by Means of Statistical Hypothesis Tests and Metamorphic Testing. Quality Software, 2007. QSIC ’07. Seventh International Conference on Quality Software (2007), p. 404-409
32. Join Methods (DB2). http://publib.boulder.ibm.com/infocenter/db2luw/v9r7/index.jsp?topic=%2Fcom.ibm.db2.luw.admin.perf.doc%2Fdoc%2Fc0005311.html (22.03.2012)

Acknowledgements

I would like to thank the IBM Deutschland Research & Development GmbH in Böblingen for giving me the opportunity to write my Diploma thesis about a very interesting topic. In particular Ingo Hotz PhD, Andrea Kapahnke, Joanna Janusz, and Pawel Koperek who have been supporting me during the whole time of the thesis.

I also would like to thank Holger Schwarz PhD of the University of Stuttgart for his support and very good and constructive comments.

Erklärung / Statement

Ich versichere, dass ich diese Diplomarbeit selbständig verfasst und nur die angegebenen Hilfsmittel verwendet habe.

I assure that I have written this Diploma thesis by myself and I have only used the specified tools.

Moritz Semler