AUTOMATED PERFORMANCE TEST GENERATION AND COMPARISON FOR COMPLEX DATA STRUCTURES

Exemplified on High-Dimensional Spatio-Temporal Indices

Mathias Menninghaus

Disputation am 07. August 2018

Dissertation zur Erlangung des Doktorgrades (Dr. rer. nat.)
des Fachbereichs Mathematik/Informatik
der Universität Osnabrück

Erste Gutachterin:	Prof. Dr.-Ing. Elke Pulvermüller
Zweiter Gutachter:	Prof. Dr. rer. nat. Martin Breunig

Abstract

There exist numerous approaches to index either spatio-temporal or high-dimensional data. None of them is able to efficiently index hybrid data types, thus spatio-temporal and high-dimensional data. As the best high-dimensional indexing techniques are only able to index point-data and not *now*-relative data and the best spatio-temporal indexing techniques suffer from the *curse of dimensionality*, this thesis introduces the Spatio-Temporal Pyramid Adapter (STPA). The STPA maps spatio-temporal data on points, *now*-values on the median of the data set and indexes them with the pyramid technique. For high-dimensional and spatio-temporal index structures no generally accepted benchmark exists. Most index structures are only evaluated by custom benchmarks and compared to a tiny set of competitors. Benchmarks may be biased as a structure may be created to perform well in a certain benchmark or a benchmark does not cover a certain speciality of the investigated structures. In this thesis, the Interface Based Performance Comparison (IBPC) technique is introduced. It automatically generates test sets with a high code coverage on the system under test (SUT) on the basis of all functions defined by a certain interface which all competitors support. Every test set is performed on every SUT and the performance results are weighted by the achieved coverage and summed up. These weighted performance results are then used to compare the structures. An implementation of the IBPC, the Performance Test Automation Framework (PTAF) is compared to a classic custom benchmark, a workload generator whose parameters are optimized by a genetic algorithm and a specific PTAF alternative which incorporates the specific behavior of the systems under test. This is done for a set of two high-dimensional spatio-temporal indices and twelve variants of the R-tree. The evaluation indicates that PTAF performs at least as good as the other approaches in terms of minimal test cases with a maximized coverage. Several case studies on PTAF demonstrate its widespread abilities.

Contents

1 Introduction **1**
- 1.1 Contributions of this thesis 2
- 1.2 Organization of this thesis 2

2 Background and Related Work **5**
- 2.1 Data Types 5
  - 2.1.1 Spatial Data 5
  - 2.1.2 Levels of Detail 6
  - 2.1.3 Spatio-temporal Data 10
  - 2.1.4 High-Dimensional Spatio-Temporal Data 10
- 2.2 Index Structures and Access Methods 10
  - 2.2.1 Basic Single-value and Spatial Access Methods 11
  - 2.2.2 Spatio-Temporal Access Methods 14
  - 2.2.3 High-Dimensional Access Methods 16
- 2.3 Definitions and Requirements of Benchmarks 18
- 2.4 Benchmarking Index Structures 20
- 2.5 Performance Measurement at the Code Level 21
- 2.6 Performance Testing in Java 22
- 2.7 Software Performance Engineering 24
- 2.8 Software Testing 25
- 2.9 Control Flow Terminology and Coverage 26
- 2.10 Code Coverage in Java 30
- 2.11 Automatic Test Generation 32
  - 2.11.1 Random Testing 34
  - 2.11.2 Concolic Testing 35
  - 2.11.3 Model Based Testing 37
  - 2.11.4 Search Based Testing 37
  - 2.11.5 Summary 48
- 2.12 Conclusions for this Thesis 51

3 The Spatio-Temporal Pyramid Adapter **53**
- 3.1 Analysis of the X-Tree 53
- 3.2 Design of the Spatio-Temporal Pyramid Adapter 55
- 3.3 Querying 58
  - 3.3.1 (1) Identify Interval Query Types 59
  - 3.3.2 (2 + 3) Conversion to Region Queries 60

3.3.3 (4) Adapt to *now*-relative Data . . . 61
3.3.4 (5 + 6) Query the Pyramid Space . . . 64
3.3.5 Query Alternatives . . . 64
3.4 Implementation of the STPA . . . 64
3.5 Classic Workload Evaluation . . . 67
3.5.1 Workload Generator and Setup . . . 67
3.5.2 Results . . . 70
3.6 Conclusions . . . 74

4 Automated Performance Comparison 75
4.1 Overview and General Idea . . . 76
4.1.1 Example . . . 78
4.2 Summary . . . 81

5 The Performance Test Automation Framework 83
5.1 Instrumentation and Code Coverage Measurement . . . 83
5.1.1 Instrumentation and CFG Detection . . . 85
5.1.2 Coverage computation . . . 87
5.2 Automated Test Generation . . . 91
5.2.1 Evaluating Existing Test Generators . . . 91
5.2.2 Automated Test Generation based on an Interface . . . 97
5.2.3 Optimizing the Length of Test Sets . . . 108
5.3 Alternative Test Set Generators . . . 109
5.3.1 Guided Workload Generation . . . 109
5.3.2 Specialized Test Generator . . . 110
5.4 Performance Measurement . . . 111
5.4.1 Overview . . . 113
5.4.2 Identification of the Steady State . . . 114
5.4.3 Identification of an appropriate number of measurements . . . 116

6 Evaluation 119
6.1 Coverage Evaluation for Test and Workload Generators . . . 119
6.1.1 Setup . . . 119
6.1.2 Results and Analysis . . . 123
6.2 Case Studies of the IBPC . . . 129
6.2.1 High-Dimensional Spatio-Temporal Indices . . . 129
6.2.2 GNU Trove Library . . . 131
6.3 Conclusions . . . 132

7 Summary and Future Work 135

Chapter 1

Introduction

Today's building and infrastructure projects, e.g. concerning the planning and construction of a subway system, require an efficient database system to store and query the plans and 3D models [52]. In large building projects like the mentioned subway example, often different variants of a construction plan need to be evaluated and presented to local authorities in order to satisfy different public requirements containing economical, ecological and other constraints. Each of these variants describe a complete construction model which in itself usually describes several discrete stages of the construction process. For the subway example, first the boring of the tunnels and the later installation of the actual tube is described in different steps of the construction process. The construction plans are usually displayed in several levels of detail (LODs). The general course of the subway through a city would be displayed in a much coarser LOD than the plan for the installation of the ventilation system. Nonetheless, both LODs effect each other as a change in the course of the subway effects the ventilation system, and the ventilation system may permit a certain course of the subway.

The different time domains introduced above are the planning time on the one hand and the building time on the other hand. In terms of database systems, the planning time is known as *transaction* time, i.e. the time when an object is inserted in, updated or deleted from a database. The building time is known as *valid* time, i.e. the time an object is valid in a database. If both time domains are combined in one model, this is known as the bi-temporal model [185]. As the time is always moving forward, this needs to be reflected in a database. For instance, if one does not certainly know how long a certain building exists, i.e. is valid in the database, the building is considered to exist until *now*, with *now* being the representation of the ongoing time. Strictly speaking, the value of *now* increases with the ongoing time and a query that matches *now* may produce different results when executed at different times. For instance, if one queries for the existence of a building in the future, before and after the current value of *now*, different results are returned. The query executed before the current value of *now* returns true and the query after the current value of *now* returns false. As one cannot predict the future existence of a building, this conservative approach is the most convenient [185] and should be supported by the desired database.

The objects which are stored in the database for the subway project obviously have a spatial, i.e. three-dimensional, representation. As described above, this representation is different for the different levels of detail. A three-dimensional representation for each of the different levels of detail results in a model with clearly more than ten dimensions. In terms of databases, models with more than ten dimensions are considered to be high-dimensional. As

the time is also a central requirement of the database to support the planning and construction of building projects, the model described here is a *high-dimensional spatio-temporal* model.

Efficient queries in a database require a fast and efficient index structure. Generally, an index structure arranges the data in a database such that it can be queried more efficiently. For the described spatio-temporal model for building plans the most efficient indexing technique is the R^{ST}-tree [172]. The R^{ST}-tree is not able to properly index high-dimensional data. In addition, no high-dimensional index exists which is able to store spatio-temporal data. Filling this gap, this thesis introduces a new high-dimensional spatio-temporal index structure which fits the requirements for an efficient indexing of the desired planning data.

The evaluation of such a new complex data structure can only be based on likewise new benchmarks which are explicitly designed for the given problem. Such benchmarks can be biased in two ways: They may miss certain configurations that are needed to adequately describe the given problem or a complex data structure which is evaluated by a benchmark may be designed to exploit a certain behavior of the benchmark. This thesis not only evaluates the new index structure with a new benchmark that bases on an already existing one but introduces a complete new system to automatically generate performance tests that avoid the aforementioned problems of biased benchmarks. The new system automatically creates suitable performance tests for a given set of competitors on basis of a common interface and compares them. The new system does not address a certain behavior which should be evaluated but evaluates the complete capabilities of the competitors on basis of the common interface. In comparison to the classic benchmark, the new automated system unfolds a different behavior of the evaluated high-dimensional spatio-temporal index structure. This does not mean that the existing benchmarks need to be replaced by the new system but the new system may be used in addition to the usually applied benchmarks in order to unfold the complete capabilities of the competitors most likely unbiased and automatically.

1.1 Contributions of this thesis

The contributions of this thesis are manifold. First and foremost, a new efficient high-dimensional spatio-temporal index structure, the Spatio-Temporal Pyramid Adapter (STPA) and a new automatic performance comparison system, the Interface Based Performance Comparison (IBPC) technique, are introduced. With the system for performance comparison a new approach on comparing the performance of a set of competitors on the basis of a common interface is introduced. The implementation of the IBPC, the Performance Test Automation Framework (PTAF), addresses several issues: it contains a new, easily extendable system for the computation of the control flow oriented coverage of Java programs, a new system for the automatic generation of performance tests for complex data structures in Java and a new system for the robust and reproducible measurement of the performance of Java programs.

1.2 Organization of this thesis

This thesis is organized as follows: Chapter 2 describes the background on spatio-temporal high-dimensional index structures as well as on general benchmarking and (performance) test generation and performance measurement. As the new techniques are mostly built on top of the existing ones, the background and related work are described together in one chapter. Chapter 3 describes the new Spatio-Temporal Pyramid Adapter (STPA) and compares it to

the R^{ST}-tree [172] using an adapted classic benchmark for spatio-temporal data. Chapter 4 introduces the new technique for the comparison of the performance of a set of competitors with a common interface. Chapter 5 describes the implementation of the new technique along with the new systems for coverage computation, performance test generation and performance measurement in Java. Chapter 6 evaluates this implementation and compares the new approach of performance comparison to the classic benchmark. For the demonstration of the capabilities of the system another case study on rather simple data structures is performed. Chapter 7 summarizes this thesis and gives an outlook on future tasks in the affected fields.

Chapter 2

Background and Related Work

This chapter presents the current state of the research related to this thesis. It is presented in the order of its appearance in this thesis, i.e. it begins with spatio-temporal and high-dimensional index structures and their benchmarks, is continued with the general definition of benchmarks, performance test generation and their specifics in Java programs and concluded by the basics and the current state-of-the-art in software testing and the automated generation of unit and GUI tests.

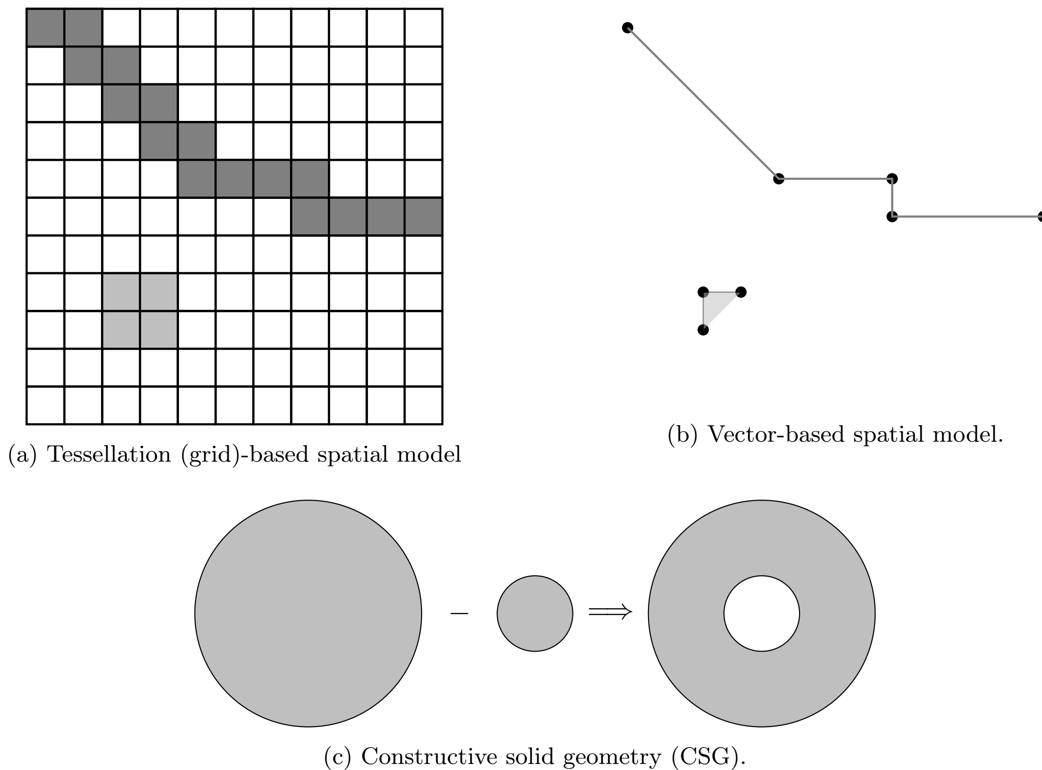
2.1 Data Types

This section gives the necessary background on high-dimensional spatio-temporal data and its indexing. Emerging from the field of Geo Information (GI) Science [79], the data types outlined here are not the only ones to consider. In addition, as the main requirement behind the new indexing technique described in chapter 3 is not only the support of spatial data as used by GI Systems (GIS) but also general index support for data used by Building Information Modeling (BIM) tools, only a rough overview of the general achievements in these fields is given.

2.1.1 Spatial Data

Peuquet (1984) [156] states that "Since no model or abstraction of reality can represent all aspects of reality, it is impossible to design a general-purpose data model that is equally useful in all situations.". In GIScience, two general types of spatial data models exist, tessellation-based and vector-based models [156]. Tessellation or polygonal mesh models use a simple geometric entity as the basic unit of space. Raster geometries are probably the most prominent subtype of tessellation. In a raster geometry, the basic unit of space is one square, represented by one pixel in a data file (see Figure 2.1a). Other examples of tessellation are triangular meshes and irregular models, which contain triangles of different sizes. In vector models, the point is the base of all represented information. Points may be extended to lines which may be extended to faces (see Figure 2.1b). There also exists a variety of hybrid models, containing both, tessellation and objects represented by vectors [156].

For the use in BIM tools, Constructive Solid Geometries (CSG) as provided by the Industry Foundation Classes (IFC) [121] are often used. In a CSG, the spatial representation is built up by a list of primitives and boolean operators on these primitives. Only if all primitives and operators have been computed in the correct order, the desired representation is complete.



(a) Tessellation (grid)-based spatial model

(b) Vector-based spatial model.

(c) Constructive solid geometry (CSG).

Figure 2.1: Three types of spatial models.

An example is provided in Figure 2.1c: a circle with a hole in the center is created by a circle minus another, smaller circle.

All of these models may be adapted for the higher dimensional space. For instance, in three dimensions, the squares in Figure 2.1a may be extended to cubes. The points in any vector-based approach may have more coordinates than only two, enhancing the lines and faces to polyhedra and other three-dimensional objects. The primitives in a CSG may also be extended to the three-dimensional space, for instance by replacing the circles in Figure 2.1c with spheres. The indexing techniques investigated in this thesis only require the minimum bounding rectangles of a spatial object and functions to determine the topological spatial relationship between two objects. Therefore, the concrete spatial model used for the representation of the objects is irrelevant. Egenhofer (1989) [78] defines a formal definition of the possible topological relationships between two spatial objects. This may be used for d-dimensional objects as well. A spatial object may be divided into its exterior, interior and boundary, which separates ex- and interior. The binary topological relation always relates to all combinations of boundary and interior of the two objects, resulting in eight different possibilities of relation as visualized in Figure 2.2.

2.1.2 Levels of Detail

Different spatial data types are not only required to model different aspects of the reality, but also to avoid the storage of any unnecessary data. Usually, spatial data may be generalized and

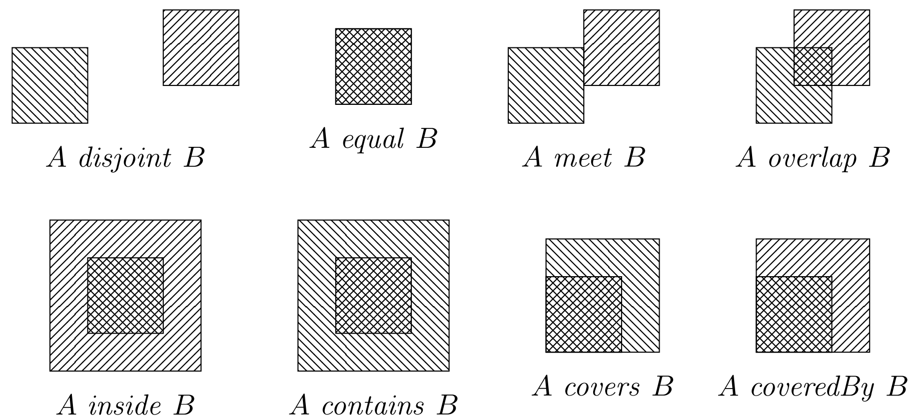


Figure 2.2: The eight topological relationships between two spatial objects according to Egenhofer (1989) [78].

details may be spared in order to save data space. In addition, objects in a more generalized spatial model can be compared to each other with less computation resources. In terms of building and city models, the original CityGML [112] standard defines five levels of detail (LOD). In contrast to the generalization of a detailed model, Borrmann et al. (2015) [50] refine a coarse representation and add more and more details with every LOD. They also focus on preserving the consistency between the LODs by using a procedural model as spatial model which is similar to a CSG. Both approaches are compared in Figure 2.3. Biljecki et al. (2016) [49] state that the CityGML standard does not define the different LODs differentiated enough and enhance it to 16 levels of detail.

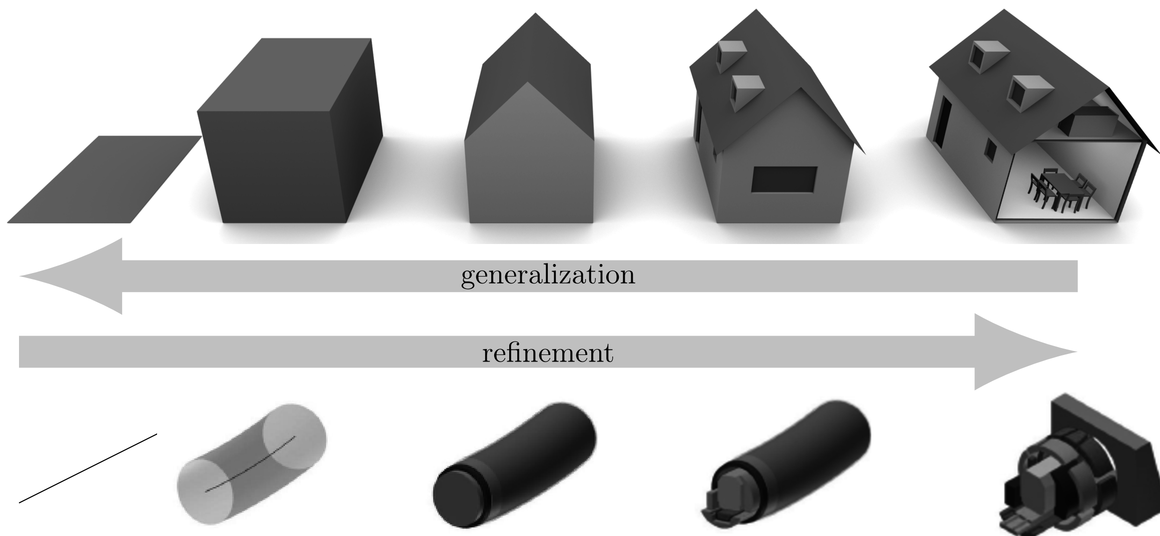


Figure 2.3: Levels of detail as described by Kolbe (2009) [112] (above) and Borrmann et al. (2015) [50] (below). The first approach generalizes the spatial data, the latter refines it.

2.1.2.1 Temporal Data

In the scope of this thesis, temporal data is not limited to the mere use of timestamps or other static time representations in databases. All time related values are linked to the ongoing real-time, called *now*. The *now*-relative data faces several difficulties as data sets which have to be stored ideally should automatically and constantly evolve after they have been stored without an update of the data sets themselves. Consider Figure 2.4 which depicts the actual employment in a software company as a Table in a relational database and the corresponding temporal model.

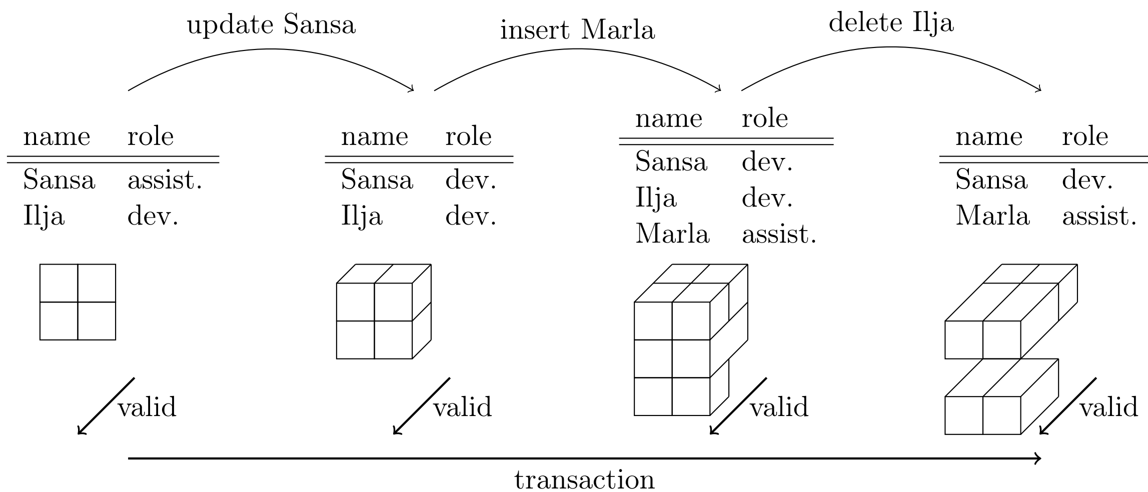


Figure 2.4: Example of a database table which changes over time (above) and the corresponding bi-temporal model (below). For visualization purposes, the abbreviations dev. for developer and assist. for assistant are used.

The bi-temporal model consists of two time axes, the valid and the transaction time. The transaction time denotes when entries are inserted and deleted physically from the database. The valid time denotes whether an entry is valid at a specific time in the modeled reality. In the given example, the entry *Sansa* is updated in the first step. In the second step, an entry of Marla is inserted. Note that the entry of Ilja has not changed on the valid time axis, it is the same entry as it has been at the beginning. In contrast, the entry of Sansa has changed. Through the transaction time, one may also note that it displays the growth and reduction of the table. In the third step, Ilja is deleted from the database, thus the transaction time of the entry ends at that point, only the entries of Sansa and Marla are continued. In each of the four pictures in the lower row, the latest entry in view of the valid time is considered to be valid and it should be valid until it changes. Snodgrass and Ahn (1985) [185] survey different approaches on handling time in databases and introduce three kinds of time: valid, transaction and user-defined time. User-defined time handles the information that is not addressed by valid and transaction time. For instance, if the update of Sansa's role takes some time, this may be modeled by the user-defined time as an offset to the valid-time. Snodgrass and Ahn (1985) [185] combine the use of valid and transaction time to (bi-)temporal databases and demonstrate the use of *now* in databases. In the example given in Figure 2.4, the visualization of valid

and transaction time does not involve *now*. Therefore, Table 2.1 denotes the information of Figure 2.4 as one bi-temporal database table.

Table 2.1: Example of a bi-temporal database table. Begin and end of the valid (VT) and transaction (TT) time are defined by $\vdash$ and $\dashv$, respectively.

name	role	$VT^{\vdash}$	$VT^{\dashv}$	$TT^{\vdash}$	$TT^{\dashv}$
Sansa	assist.	05/2013	05/2014	05/2013	*now*
Sansa	dev.	05/2014	*now*	05/2014	*now*
Ilja	dev.	05/2013	05/2015	05/2013	05/2016
Marla	assist.	05/2015	*now*	05/2015	*now*

It should be noted that transaction and valid time may not be linked to each other. For instance, if a plan for the future is made, all operations on the transaction time already may have been fulfilled before the valid time of the first entry begins. Also, all entries may have ended on valid time before the first operation on the transaction time begins. For instance, if an old archive is stored in a newly set up database or if historical data is stored. Thus, as the end value in the valid and transaction time may either be a concrete value or *now*, four different types for the combination of valid and transaction time exist in the bi-temporal model. Once again, it must be clearly distinguished between a change in the modeled reality, e.g. the promotion of Sansa, and a change on the database, e.g. the actual update of Sansa's entry. It is always a part of the modeling process if and how to set transaction and valid time in relation to each other.

Clifford et al. (1997) [66] discuss the semantics of *now* in more detail and provide a framework for working with bi-temporal data. In the literature, *now* is also denoted as ∞, -, @ and *until-changed* [66]. When incorporating *now*-relative data, one must always consider that the concrete value of *now* changes over time and *now* has to be modeled properly. To exemplify one of these problems imagine that *now* is modeled as ∞ or a very high value, known as the optimistic approach. Querying if Sansa is still an assistant at 06/2014 would deliver true, if queried before 05/2014 and false if queried after 05/2014, when the first entry of Sansa is updated with her promotion. Following this example, it always must be considered that as abstract *now* may be, the current value of *now* always is clearly defined for one certain time in the database and may affect queries, updates, deletions and insertions.

Aside from the semantics of *now* and the way it is modeled, the representation of *now* in the database has to be chosen. Stantic et al. (2009) [186] distinguish between four approaches to represent *now*, which are called NULL, MIN, MAX [190] and POINT. The concept of the NULL approach is that *now* is represented by a value outside of the order of the temporal domain. The MIN and MAX approaches use the minimum and maximum values, e.g. $-\infty$ and ∞, of the temporal domain. The POINT approach models *now* with the same value as the beginning of transaction and valid time, respectively. For instance, the value for $VT^{\dashv}$ for Marla in Table 2.1 would be set to 05/2015 if the POINT approach is used. Stantic et al. (2009) [186] claim that their POINT approach outperforms MAX and the other approaches when applied on a database with the B^+-tree or R^+-tree as indexing technique. Anselma et al. (2013) [35] introduce the NOT-NOW approach which excludes the usage of *now* and always assumes that the ending times of both, valid and transaction time are known for all objects. They use this approach to compare the cost of *now*-relative data to temporal data without *now*.

The possible topological relationships between temporal intervals are defined by Allen (1983) [31], refined by Kriegel et al. (2001) [116] and visualized in Figure 3.3 on page 59.

2.1.3 Spatio-temporal Data

In contrast to the isolated perspective on exclusively spatial or temporal extends of objects, spatio-temporal data combines both into one data model. Worboys (1994) [208] introduces spatio-bi-temporal data, consisting of two spatial and two temporal (transaction and valid time) dimensions. Along with examples on administrative areas, road networks and land ownership an incomplete inside on possible topological spatio-temporal relationships is given. As *now* is considered to be an increasing value within the spatio-temporal data space, the usual topological spatial relationships as defined by Egenhofer (1989) [78] (see 2.2) may be applied, adopted by the temporal interval relationships by Allen (1983) [31] (see 3.3). He et al. (2013) [97] combine the 13 interval relationships to the fundamental eight binary relationships between d-dimensional objects as depicted in Table 3.1 on page 59.

Speaking of spatio-temporal data, one must clearly distinguish between discretely changing and continuously changing spatio-temporal data. Discretely changing data like the road-networks, administrative areas or land ownerships exemplified in [208] only changes at concrete time stamps. Continuously changing spatio-temporal data models are not discretely changing but moving through space and time. For instance, modeling the users of mobile networks within a grid of antennas is one possible application of a spatio-temporal model. Querying those models would always imply that if the query targets the space between two data points, the actual query result is the result of an interpolation between those two data points. In this thesis it is not dealt with moving objects but with non-moving yet discretely changing data.

2.1.4 High-Dimensional Spatio-Temporal Data

Given the general d-dimensional definition of spatial data, the possibility of more than two temporal dimensions which are linked to the ongoing *now*-value and the need for up to 16 levels of details, a data model easily becomes high-dimensional, speaking of more than 10 dimensions. Those dimensions do not need to be spatial or temporal dimensions exclusively, but also thematic and therefore may not only be represented by an interval in one dimension but by a single value of an ordered set. Nonetheless, all dimensions may be modeled within the possibilities of the general spatio-temporal approaches outlined in the sections above.

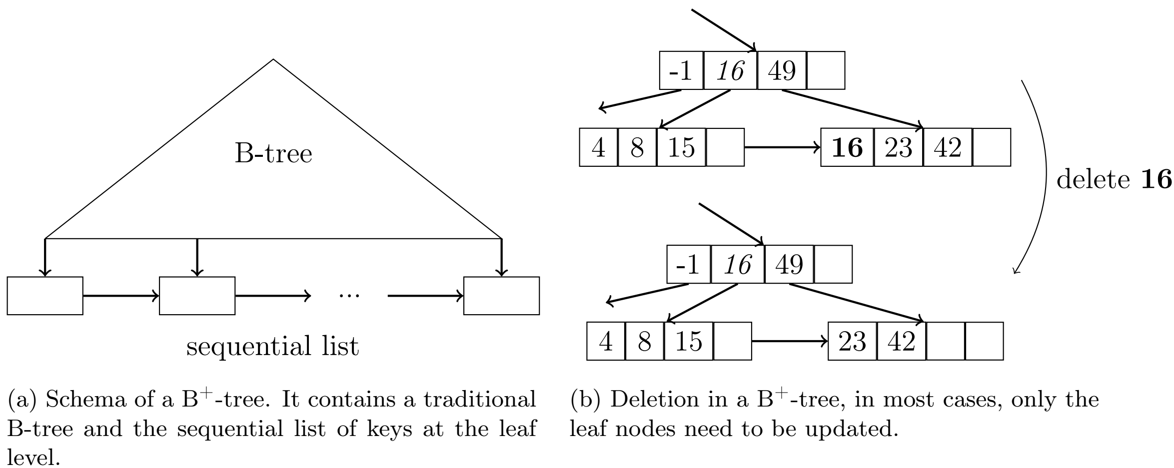
2.2 Index Structures and Access Methods

In this section, all access methods related to the technique developed in this thesis and their basics are described. Starting with the basic access methods for the indexing of large sets of single values or spatial (2D and 3D) objects and continuing with specialized access methods for the retrieval of spatio-temporal and high-dimensional data. Generally speaking, each of the described techniques tends to split the data space into sub spaces and recursively splits those sub spaces again until a set of small sets of objects exists which may be accessed much faster than by sequentially scanning each object in the given data set. The smallest subset of objects should ideally fit onto one hard disk block to achieve the most efficient number of accesses. Thus, indexing usually describes how the data is ordered, or indexed, on the hard disk or in memory and the access method ensures the ideal use of such an index. Both may

often only be used in combination and the literature uses the terms *index* and *access method* interchangeable.

2.2.1 Basic Single-value and Spatial Access Methods

This section focuses on those access methods which are enhanced or used by spatio-temporal and high-dimensional access methods and are initially designed to grant efficient access on single values and spatial data (2D and 3D).



(a) Schema of a B$^+$-tree. It contains a traditional B-tree and the sequential list of keys at the leaf level.

(b) Deletion in a B$^+$-tree, in most cases, only the leaf nodes need to be updated.

Figure 2.5: Basic concepts of a B$^+$-tree according to [67].

The B-tree [43] indexes single values as a tree structure. Every node holds between k and $2k$ keys, except the root which may hold between 1 and $2k$ keys. The keys are ordered from left to right in each node and each key in a non-leaf node points to a node which contains keys that are less than the key in the parent node. The retrieval of a key means to traverse downward from the root until the key is found or a leaf which does not contain the searched key is reached. Inserting in a B-tree may result in a node which contains more than $2k$ keys, such a node must be split up into two nodes with at least k keys. As the parent node gains one key, it may also be split up recursively. During deletion, which may result in nodes with less than k keys, nodes must be merged upwards recursively. Several enhancements of the B-tree exist. The most popular is probably the B$^+$-tree. In a B$^+$-tree [67], the key set and the tree structure are separated from each other. On the leaf level, all keys are stored in an ordered sequential list of leaf nodes, with the leaf nodes being linked together from left to right. Doing so, most of the time only the leaf nodes need to be updated. The non-leaf nodes only need to be updated when a leaf node has to be split up or merged. Additionally, range queries become much faster. One only has to traverse to the leaf where the nearest start point of the result set is located and then sequentially go through all following keys until the first key outside the query range is reached. Figure 2.5 displays the main properties of the B$^+$-tree. Instead of inserting or deleting one key at time, B-trees may be built up more efficiently by using bulk-loading techniques [90], which may be used to insert a set of keys and constructing a B-tree with an efficient load. An efficient load of the B-tree nodes is achieved when the *usual* number of operations can be performed without fragmentation of the tree. A B-tree which is

able to store new keys without a split after almost every insert and which is able to delete keys without merging after almost every deletion, is considered to be more efficient. Also, B-trees with a high density of keys per node are more efficient in lookup and query operations.

The space-partitioning Quadtree [80] is initially designed for two-dimensional objects, but may be extended easily to higher dimensions, for instance by the Octree for three-dimensional data [130]. In a Quadtree, every non-leaf node has four children, pointing to child nodes. One node always splits the space vertically and horizontally with the orthogonal intersection of both splitting axis being the node itself. That is, the partition in sub spaces does not need to be symmetric. The creation of a new node only splits the data space within the sub space in which the new node is created. Insertion, deletion and search work analogously to binary approaches. The Grid File [144] partitions the multidimensional data space by a uniform grid of buckets. These buckets are accessed by a directory file in order to ensure the two-disk access principle for point queries. A bucket that contains more entries than a data block is able to hold is split, two neighboring buckets which would fit into one are merged. As the Quadtree, the Grid File may be extended to more than two dimensions. For indexing point data, the K-D-B-tree [162] partitions the data space into point pages and merges them into region pages. These region pages are recursively merged up to a root page which represents the whole data space. The K-D-B-tree [162] partitions the data space by alternating the split axis. That is, in the two dimensional case, a sub space which has been created by splitting along the x-axis will be split along the y-axis.

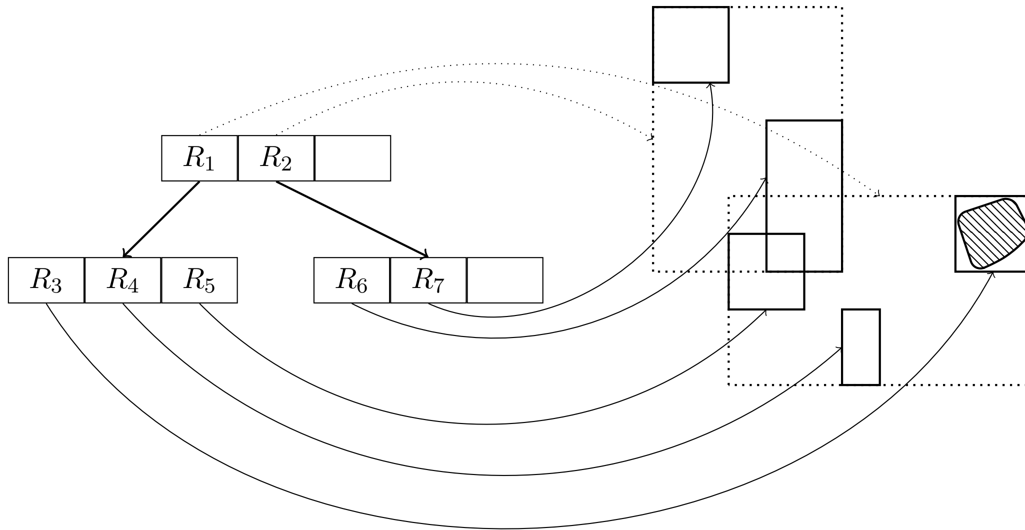


Figure 2.6: Structure of an R-tree [95]. The ruled area is the example of a spatial object surrounded by its minimum bounding rectangle (MBR). The entries in one node with a maximum capacity of three point to sub-nodes.

According to Van Oosterom et al. (2002) [194], the most efficient technique for the storage and retrieval of two and three dimensional spatial data is the R^*-tree [44]. The R^*-tree is based on the R-tree [95], which is exemplified in Figure 2.6. An R-tree consists of entries and nodes. An entry contains a minimum bounding rectangle and a pointer to a node. A

node consists of a set of entries. The minimum bounding rectangle (MBR) in an entry is the minimum enclosure of the spatial extend of all entries in the corresponding node. The entries in the leaf nodes contain the spatial objects which are stored by the tree. All nodes, except for the root, contain between m and M nodes.

New entries are inserted by traversing from the root through those nodes which would enlarge least by including the new entry and adding them at the leaf level. If the chosen leaf node contains more than M entries after insertion, it is split into two new nodes and the split is propagated upwards in the tree if necessary. In the original variant of the R-tree, three split variants are proposed: the exhaustive, the linear and the quadratic split. The exhaustive variant tries all possible groupings and chooses the best, i.e. the tuple of entry sets whose MBRs cover the search space least. Doing so, future insertions will affect the existing nodes least. With the node to be split containing $M+1$ entries, approximately 2^{M-1} groupings exist to create the desired tuple.

The quadratic algorithm picks two entries to be the seeds for the desired two new groups of entries. The two seeds E_i, E_j are chosen such that they waste the most space $a = area(E_i \bigcup E_j) - area(E_i) - area(E_j)$ with *area* being a function which calculates the covering area (2D) or volume ($\geq$3D) of the entry's MBR. From the group of remaining entries the entry which has the maximum difference in area enlargement between the two groups is chosen if it is included by the entries. The chosen entry is added to that group which has the least area enlargement by including the chosen entry. The quadratic algorithm has quadratic cost in the number of maximum entries and linear cost in the number of dimensions [95].

The linear algorithm works similar to the quadratic algorithm. The seeds of the two groups are chosen as follows: For each dimension, the MBR with the maximum start value and the one with minimum end value are chosen. The difference between these two rectangles is divided by the width of the entire set of entries to be split along the dimension according to which the rectangles have been chosen. At last, the entries which have the greatest relative separation along any dimension are chosen as seeds. The algorithm then chooses one of the remaining entries at a time and assigns it to that group which has the least area enlargement by including the chosen entry [95].

When deleting entries from an R-tree, a node may become underful, i.e. containing $< m$ entries. The remaining entries are then also deleted and the deletion is propagated upwards. All entries which have been deleted that way are reinserted at their corresponding level in the tree afterwards.

Instead of only taking the area and its enlargement into account for finding the best distribution of entries in an R-tree, the R*-tree also incorporates the margin and overlap of the p MBRs, with an overlap being defined as [44]:

$$overlap(E_k) = \sum_{i=1, i \neq k}^{p} area(E_k \bigcap E_i), 1 \leq k \leq p \tag{2.1}$$

The insertion algorithm in the R*-tree is altered in contrast to the R-tree. If, via traversing the tree, a node is reached which points to leaf nodes, the node with the least overlap enlargement to include the new entry is chosen. The R*-tree also introduces an alternative to the immediate split of full nodes which is called forced reinsert. During the insertion of a new entry, if a node is full, p of the entries of the full node are reinserted at the corresponding level instead of performing a split of the complete node. Before reinsertion, all $M+1$ entries of a node N are sorted decreasingly by their distances to the center of the MBR enclosing N.

Either the p entries with the maximum or minimum distance may be chosen for reinsertion which is called far reinsert and close reinsert respectively. The reinsertion of a node is only performed once per node level and insert operation. If a node, at a level at which a reinsert already has taken place, is full, it is split as usual.

Beckmann et al. (1990) [44] also propose another split algorithm, which is called R*-split for the remainder of this thesis. All $M+1$ entries are first sorted by their starting values and then sorted by their ending values for each dimension. For each of the sorts, $M-2m+2$ distributions into two groups are investigated. Given $k=(1,..,M-2m+2)$ the first group of the k-th distribution contains the first $m-1+k$ entries and the corresponding second group contains the remaining entries. In order to choose the split axis for each dimension, the sums of all margin-values of the two groups of the different distributions are computed. The margin value of a distribution is the margin of the bounding box of the first group plus the margin of the bounding box of the second group. The axis which has the minimum of all those sums is chosen as split axis. Along that axis, the distribution with the minimum overlap value is chosen. The overlap value is the area of the intersection of the bounding box of the first group with the bounding box of the second group. Ties are resolved by choosing the area which has the minimum of the sum of the areas of the two groups.

Note that both, the R-tree and the R*-tree, may be built completely up differently when the same set of entries is inserted in another order. This behavior is intensified if deletions are allowed [44].

Hellerstein et al. (1995) [99] generalize the basic principles of the B-tree and R-tree and their variants to the Generalized Search Tree (GiST). Although the generalization makes a sub-class of indices interchangeable, the specifics of that approach make the implementation of index structures that chose different approaches than R- and B-tree difficult and make index-specific optimizations more complicated. Analogously, Aref and Ilyas (2001) [37] define an extensible database index for the support of space partitioning trees like the Quadtree.

2.2.2 Spatio-Temporal Access Methods

Most spatio-temporal access methods are designed to handle continuously moving objects. Extensive surveys on access methods for moving objects can be found in [27, 138, 143, 155, 26]. In this thesis, it is dealt with non-moving, yet discretely changing spatial data. Index structures which support this type of data are presented in this section. In contrast to pure spatial data, the index structures have to deal with growing rectangles if the end value is set to *now* on at least one of the temporal dimensions. It is important to note that if one leaf-entry contains a *now*-relative rectangle, all nodes which include that *now*-relative rectangle must also be *now*-relative.

Spatio-temporal structures are mainly derived from existing spatial index structures. For instance, the Quadtree [80] is extended by Tzouramanis et al. (1998) [192] to the Overlapping Linear Quadtree that is able to track the evolution of raster images on the transaction time. Or the B-tree [43] which is extended to the B^{X}-tree by Jensen et al. (2004) [107].

As the R*-tree [44] is considered to be the most efficient method for indexing spatial data, it is the most often used technique to be adopted by spatio-temporal indexing techniques. The improvements on the R-tree regarding spatio-temporal data can be separated into two branches.

First, there are indexing methods that combine several R-trees in order to take the *now*-relative temporal dimensions into account. The Historical R-tree [140] is an R-tree of R-trees,

one for every time step, where new R-trees only store changed objects and use references to the subtrees of the unchanged nodes in the previous R-tree in order to save space. The 2+3 R-tree [141] uses one 2-dimensional R-tree to store the current spatial information and one 3-dimensional R-tree to store all past data, i.e. every object that already has *now*-relative temporal intervals. If every state of the object is known a priori, the 2+3 R-tree is reduced to a 3-dimensional R-tree. Both approaches are only able to handle one temporal dimension.

Secondly, some indices try to enhance the R-tree with spatio-temporal functionality by changing its insert, split and delete algorithms. The R^{ST}-tree by Saltenis and Jensen (1999) [172] uses time-parametrized values and an additional split algorithm to take growing rectangles in the valid- and transaction-time into account. Time parametrization means that by every calculation involving the *now*-value *now* is replaced by the current value of *now* plus a data-dependent (large) parametrization value. Doing so, the computation may foreshadow the future expansion of the spatio-temporal objects along their temporal dimensions. Additionally, the user may prioritize the spatial or the temporal component. For instance, the volume of the spatio-temporal MBR r is computed with:

$$volume(r) = \begin{cases} bitemporalarea(r)^{1+\alpha} \cdot spatialarea(r) & \text{if } \alpha \leq 0 \\ bitemporalarea(r) \cdot spatialarea(r)^{1-a} & \text{otherwise} \end{cases} \tag{2.2}$$

Where $\alpha \in [-1, 1]$ is the user defined value which indicates how to prioritize the spatial or temporal component. Note that the margins and overlaps are computed similarly in the R^{ST}-tree. As always, $spatialarea(r)$ is considered to be the volume of r if r is an object in the 3D or higher dimensional space.

The split algorithm of the R^{ST} compares the distribution chosen by the R^*-split algorithm to an additional distribution and choses that one with the minimum overlap value, ties are resolved by choosing the distribution with the minimum area value. The R^{ST} distribution is generated with the general goal to separate *now*-relative and non *now*-relative rectangles. The approach distinguishes three types of rectangles. Static regions (□) are those with certain end values in the transaction time. Note that so-called static stair-shapes are rectangles whose end values are *now* for the valid time and are considered to be static. Growing rectangles (⮺) are those rectangles whose end value is *now* for the transaction time only and the end values of growing stair-shapes (⦣) are *now* for both, the valid and the transaction time. During a split, the two resulting groups are set to the type following Table 2.2. Saltenis and Jensen (1999) [172] state that the R^{ST}-split distribution is used in 19% of the cases and improves the query performance between 10% and 25%.

The forced reinsert of the R^*-tree is altered. It also reinserts one entry at a time up to p entries but reinserts that entry which, when removed, shrinks the volume of the MBR of the overfull node the most.

One extension of the often-cited Time-Parametrized-R-tree (TPR-tree) [174], the R^{EXP}-tree [173], takes moving objects into account by simply using integrals for the R^*-tree operations such as *union*, *overlap*, *volume* and *margin* which denotes the length of the boundaries of a rectangle. Doing so, it is able to build up conservative minimum bounding rectangles (MBR) which take the future extension of the spatio-temporal MBRs into account. It is not necessary to update the MBRs with every change of the object's shape and position. The R^{EXP}-tree does not use a constant parametrization value but a dynamical computation via a time horizon function. Every n insertions, the time duration Δt of the last n insertions is computed, where n equates to the number of entries in a node. Then the update interval length is approximated

Table 2.2: Prioritization for the types used for two resulting groups of the R^{ST}-tree split distribution according to Saltenis and Jensen (1999) [172]. With $k = M + 1 - m$ where M is the maximum and m is the minimum number of entries per node.

priority	first group	second group	enabling condition
1	□	□	$|↳| + |∡| = 0$
2	□	↳	$|∡| = 0 \wedge 0 < |↳| \leq k$
3	↳	↳	$|∡| = 0 + |↳| > k$
4	□	∡	$|∡| > 0 \wedge |∡| + |↳| <= k$
5	↳	∡	$0 < |∡| \leq k \wedge |↳| + |∡| > k$
6	∡	∡	$|∡| > k$

as $UI = \left(\frac{\Delta t}{n}\right) N$, where N is the number of leaf entries and the querying window length as $W = \alpha_W \cdot UI$ with $0 < \alpha_W < 1$. The time horizon H is computed as $H = W + UI$. The time horizon function can also be used to enhance the R^{ST}-tree. As there seems to be no other indexing method which is designed for discretely changing spatial and bi-temporal data, the R^{ST}-tree is the most suitable competitor to the technique developed in this thesis.

Stantic et al. (2010) [187] propose a new indexing technique for temporal data based on the relationships between intervals [31], the TD-tree. Despite the fact that this technique does not incorporate *now*-relative data, it is used by He et al. (2013) [97] to create a parallel indexing technique for spatio-temporal data. Although it does not support *now*-relative data, it provides another perspective on indexing high-dimensional spatial data which is partly used in this thesis.

2.2.3 High-Dimensional Access Methods

The R-tree [95] seems to be the best choice not only for indexing spatial but also point and spatio-temporal data. Nonetheless, the performance of the R-tree and its best known enhancement, the R^*-tree [44], decrease rapidly when used for objects with a higher number of dimensions (> 10). Berchtold et al. (1996) [46] state, that the overlap of the directory nodes of an R^*-tree increases rapidly for uniformly distributed points with increasing dimensionality. They propose the X-tree, which extends the R^*-tree with a new splitting technique and super-nodes. The split algorithm of the X-tree first tries a topological split, such like the split algorithm of the R^*-tree and then a minimum overlap split. The minimum overlap split uses the recorded split history of the index in order to find the split axis which contains an overlap free or at least overlap minimal split. The threshold in which an overlap is considered to be minimal depends on several system dependent parameters like the data page access time or the CPU time which is necessary to process a data block. If both split algorithms are not able to find a minimal overlap split, the node is extended to a super-node by enhancing the size of that node by one block.

The maximum number of entries in one node of the R-tree or a similar hierarchical structure decreases with an increasing number of dimensions. Thus, more nodes are needed to index high-dimensional data and therefore more nodes and blocks are accessed when querying the structure. Referring to that, the TV-tree [123] reduces the number of dimensions by using a telescoping function. Lin et al. (1994) [123] state that the number of dimensions used in the directory nodes to discriminate the path to the leafs is reduced significantly.

Space-partitioning methods do not face the problem of overlapping regions but the number of partitions grows exponentially [201] with an increasing number of dimensions which also causes decreasing performance. Facing these problems, Weber et al. (1998) [201] propose the VA (*vector-approximation*) file which divides the data space into 2^b rectangular cells, where b is a user defined number of bits. Together with a formula for approximatively addressing each data point and a filtering function for efficiently excluding data cells when querying, the VA-file outperforms the X- and R-tree and works even better in higher dimensions.

Beside R-tree-based and space-partition-based methods, dimension-reducing methods use to map the d-dimensional data points or rectangles onto a one-dimensional value and store these with a B^+-tree [67] or similar method. iDistance [105] identifies the d-dimensional points by the nearest reference point and the distance to this reference point. The PL-tree [200] uses a scaling function to map a real vector to an integral vector and the bijective cantor pairing function to map these d-dimensional data points into a scalar. It outperforms the X- and R-tree but is outperformed by iDistance in terms of query performance.

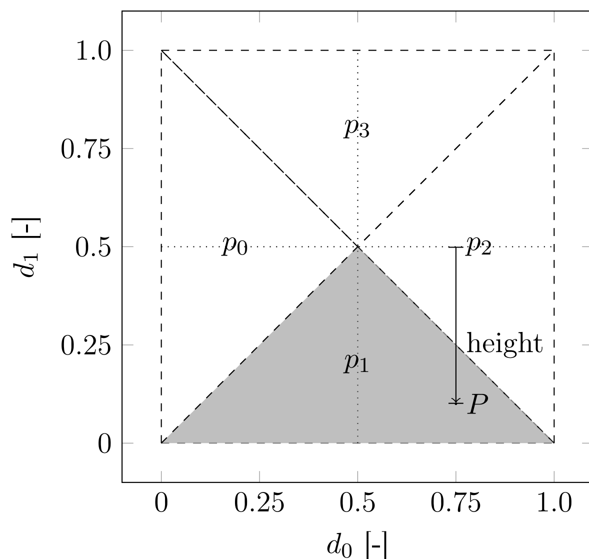


Figure 2.7: Calculation of the pyramid value of a 2-dimensional point $P(0.75, 0.1)$. The data space is divided into 4 pyramids. The pyramid value for P is 1.4: 1 because it lies in pyramid 1 plus .4 because this is its distance (height) to the center (0.5) in dimension d_1 [47].

The Pyramid Technique [47] maps d-dimensional points into one-dimensional values. It therefore splits the d-dimensional data space into $2d$ d-dimensional pyramids, whose bases are the borders and whose centers are the center of the data space. The pyramid value is calculated as follows (Figure 2.7): the places before the decimal point depict the pyramid in which the point lies and the decimal places are the height in that pyramid. The height of a point in pyramid p is the distance of that point from the center in dimension $p\ MOD\ d$. After mapping all points to their pyramid values, they are stored in a B^+-tree [67], using the pyramid values as keys, while the leaf nodes contain the original d-dimensional key. As the calculation of the pyramid value is not bijective, a mapped range query on data stored with the pyramid technique may result in more elements than expected. Therefore, one has to test every match against the original query. Moreover, a n-dimensional range query is converted into up to $2d$

one-dimensional queries because every pyramid which intersects the query rectangle has to be queried. The main drawback of the Pyramid Technique is that one has certainly to know the borders of the data space and rebuild the complete index if data is stored which lies outside the originally assumed borders. This rebuild causes an overhead of node accesses. Berchtold et al. (1998) [47] state that this overhead becomes negligible for high-dimensional data.

The Pyramid Technique can be enhanced to the Extended Pyramid Technique [47] by shifting the center of the pyramids to the median of the data set in every dimension. Doing so, the efficiency of querying on clustered data sets is improved, but the center of the pyramids needs to be altered if its distance to the real median is to high. Therefore, an approximation of the real median of the already inserted data is tracked by a histogram and constantly compared to the actual center. Altering the center to the median requires a complete rebuild of the structure which can be done most efficiently by using bulk loading techniques on the underlying B^+-tree. Zhang et al. (2004) [212] generalize this technique to the P^+-tree which dynamically divides the data space into several subspaces in order to deal with more than one cluster of data points.

2.3 Definitions and Requirements of Benchmarks

In order to compare the new spatio-temporal high-dimensional indexing technique developed in this thesis (Chapter 3) to existing approaches, this section presents the background on benchmarks in general. In the current research on benchmarks and benchmark generation no general accepted definition of a benchmark can be found. Most notably, Jain (1991) [106] describes workload generation and selection as an art where every task of comparing the performance of several programs must be undertaken individually and with respect to the actual requirements. Nonetheless, the requirements for a *good* benchmark seem to be comparable to the benchmarks and benchmark generators investigated throughout this thesis. In Section 2.4 the evaluations and benchmarks for spatial, spatio-temporal and high-dimensional benchmarks are described as examples of unaudited and not verified open source benchmarks. Examples of audited and verifiable industry standard benchmarks are the following:

- The Business Applications Performance Corporation (BAPCo) [1] provides benchmarks for personal computers, tablets etc.
- The Embedded Microprocessor Benchmark Consortium (EEMBC) [4] provides benchmarks for embedded systems, like the synthetic CoreMark [3].
- The Storage Performance Council (SPC) [18] provides benchmarks for storage, such as hard disk drives (HDD) or solid state drives (SSD).
- The Transaction Processing Performance Council (TPC) [22] provides data-centric benchmarks, i.e. benchmarks for the evaluation of transaction processing and databases.
- The Standard Performance Evaluation Corporation (SPEC) [19] provides a variety of software benchmarks for single scenarios to a full system scale.

As the TPC and SPEC annually host scientific conferences, the research on benchmarks is concentrated on these two. For the TPC view on benchmarks, Huppler (2009) [104] gives a subjective overview of the current state of benchmarking and especially the TPC-C benchmark. According to [104] a good benchmark should be

Relevant The user acknowledges that the benchmark reflects an important aspect of the system under test.

Repeatable Several executions of the benchmark are independent of one another and have the same result.

Fair The benchmark does not favor a single solution or environment.

Verifiable The results of the benchmark are verified.

Economical Such that the user can afford to run the benchmark.

It is concluded that most benchmarks satisfy four of these criteria and the fifth must be "given up". Contrary, the TPC-C benchmark seems to satisfy all criteria and therefore blocks the introduction of new benchmarks. Also with focus on TPC, Patel (2015) [154] highlights three key problems in current data-centric benchmarks. These are the ambiguity of benchmarks, for instance benchmarking the runtime and energy consumption at the same time, the isolated use of benchmarks by a single vendor only and the generation of benchmarks by vendors and not by the customers, i.e. users of the benchmark. The users of a benchmark shall certainly have the clearest perception of what to measure and therefore may be able to create benchmarks which satisfy their specific needs.

Sim et al. (2003) [182] postulate the usage of benchmarks to improve the research in software engineering. Along with that, they provide a general definition of benchmarks:

Definition 2.1 *We define a benchmark as a test or set of tests used to compare the performance of alternative tools or techniques. A benchmark has three components. [182]*

These components are a "motivating comparison" which is comparable to the relevance criteria in [104], the "task example" which should contain a representative sample from the problem population and the "performance measures" or results of the benchmark. Sim et al. (2003) [182] compare benchmarking as an empirical method to experiments and case studies. An advantage of experiments is the possibility to directly compare the results but they may not be used for explanatory studies, whereas case studies are flexible and robust but the limited control may reduce the generalizability of the results. According to Sim et al. (2003) [182] a benchmark shares the advantages of these methods and successful benchmarks share accessibility, affordability, clarity, relevance, solvability, portability, and scalability as properties.

With focus on SPEC, von Kistowski et al. (2015) [198] provide the following definition of a benchmark:

Definition 2.2 *[A benchmark is a] Standard tool for the competitive evaluation and comparison of competing systems or components according to specific characteristics, such as performance, dependability, or security. [198]*

They distinguish between two types of benchmarks. Specification based benchmarks simulate a certain business problem to be solved by the competitors. For example, a specification based benchmark for spatial index structures contains workloads that address the key requirements of a spatial index structure like insertion, deletion and query of spatial objects. Kit-based benchmarks also provide the solution for further measurement. For the example of spatial index structures, a kit-based benchmark would also contain the implementation of a certain index structure and the user may alter such a benchmark only by a set of parameters. [198] also

describe five key properties of good workloads: relevance, reproducibility, fairness, verifiability and usability.

Except for the usability, these requirements equal the characteristics outlined by Huppler (2009) [104]. Where [104] only focuses on monetary characteristics, [198] also include the technical ability of the user to run the benchmark which includes an accurate description of the system requirements. The properties relevance and usability reflect the seven properties postulated by [182]. The definition in [182] emerges from the need of a better measurement of software engineering techniques and although it is often used[1], it lacks key requirements as reproducibility/repeatability, fairness and verifiability. As von Kistowski et al. (2015) [198] provide a more verified and solid approach on their definition (Definition 2.2) of a benchmark and only slightly alter that of [104], it will be used as the only definition of a benchmark throughout this thesis. Sim et al. (2003) [182] describe a benchmark as a test or set of tests, whereas [198] use the term workload(s) for the basic components of a benchmark. In this thesis, a test is a concrete set of operations performed by a competitor and then measured by the benchmark. A workload is the abstract and parametrized description of a test set.

By the knowledge of the author no general benchmark creator exists. In contrast to unit testing where several test generators exist which do not depend on specific functional requirements of the software, a benchmark is always bound to a predefined set of functional requirements.

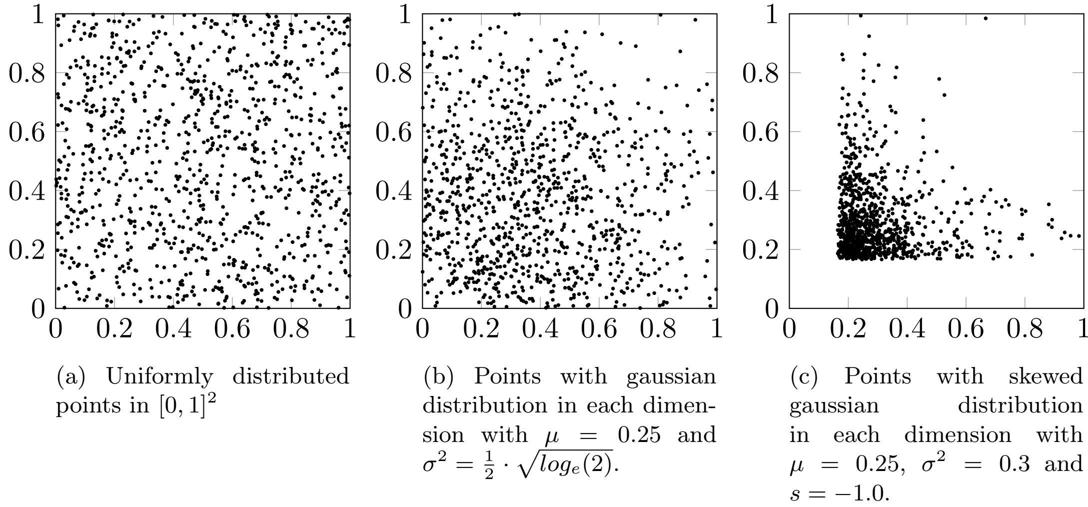
2.4 Benchmarking Index Structures

Together with high-dimensional and spatio-temporal access methods, several datasets and benchmarking applications have been proposed in order to analyze and evaluate different index structures and access methods. These approaches are described in this section.

For the generation of spatio-temporal data, which in this case means moving-objects data, the best known frameworks may be GSTD [188], OPORTO [169], and G-TERD [193]. Brinkhoff (2002) [53] proposes a framework for generating network-based moving objects e.g. traffic in road networks, just like the data created by SUMO [45]. In addition, Jensen et al. (2006) [108] (COST), Düntgen et al. (2009) [77] (BerlinMOD), and Chen et al. (2008) [64] define benchmarks for moving objects indices. None of these approaches is compared to another benchmark or verified by a comparison to real world data. The authors make suggestions on how the benchmark addresses several issues with spatio-temporal data, e.g. by incorporating several spatial distributions, but the parameters are not, for instance, created by a regression test of existing data sets or known performance goals. Also, there does not seem to exist a benchmark for the evaluation of discretely changing spatial data.

Commonly used spatial distributions are uniform, gaussian and skewed distributions as exemplified in Figure 2.8. Within uniformly distributed values, the probability for a point being located anywhere in space is equal for each coordinate within the space. A gaussian distribution is defined by a mean value μ and a variance value $\sigma^2 > 0$. A set of gaussian distributed values has a mean value of μ and 50 % of all values are located within a maximum distance of 0.675σ around μ. The density of points reduces with increasing distance to μ. That is, 90 % of all values have a distance of 1.645σ to μ and 99 % have a distance of 2.576σ to μ. A uniformly distributed value u would therefore be altered to $g = \mu + \sigma^2 \cdot u$. A skewed distribution incorporates skewness s into a gaussian distribution. If $s > 0$ the distribution is

[1] 245 citations according to Google Scholar May, 4th 2018



(a) Uniformly distributed points in $[0,1]^2$

(b) Points with gaussian distribution in each dimension with $\mu = 0.25$ and $\sigma^2 = \frac{1}{2} \cdot \sqrt{log_e(2)}$.

(c) Points with skewed gaussian distribution in each dimension with $\mu = 0.25$, $\sigma^2 = 0.3$ and $s = -1.0$.

Figure 2.8: Different distributions in a two-dimensional space.

skewed to the right and for $s < 0$ it is skewed to the left. For instance, given a uniformly distributed value u, the skewness would be applied by $y = \frac{1-e^{-s \cdot y}}{s}$ for $s \neq 0$ and as before $g = \mu + \sigma^2 \cdot y$ in order to get a skewed gaussian distribution. For $s = 0$ the distribution would remain symmetric.

The most recent approach for the analysis of indexing techniques for high-dimensional point data is, to the knowledge of the author, QuEval [175, 114], a framework which can be extended with index structures, data sets, and distance metrics. Unfortunately, it is designed for high-dimensional point and not spatial or even spatio-temporal data.

2.5 Performance Measurement at the Code Level

In this thesis, the performance of the investigated structures is measured and compared in order to find the structure with the best performance. This section describes the basics on performance measurement at the code level, which is used to compare the performance of the benchmarks and test sets developed throughout this thesis.

The performance of a system may be measured on several levels, such as performance predictions on the requirements level, or load testing on the design level. For the remainder of this thesis, performance tests and performance measurements are always computed on the implementation, i.e. code level. That is, the actual performance of the classes in the system under test (SUT) is measured. Also, here, performance measurement at first means measurement of the computation time of a set of tests performed by the SUT. In addition, performance may also be measured in terms of memory or energy consumption. In the current literature, several research areas in regard to performance measurement at the code level can be depicted. First, the investigation of performance anti-patterns and performance problems at the code level [184, 92, 205, 181]. Secondly, performance measurement of software using benchmarks [38, 54, 48], i.e. verifying whether the system meets its requirements. Thirdly, the performance measurement of different software versions using unit tests [103, 161, 55]. Existing

automated performance test generators are currently all linked to one of the mentioned research areas [92, 181, 103]. A recent literature review on performance tests in general is given by Freitas and Vieira (2014) [84].

In contrast to the well-known software patterns [85], anti-patterns [184] can be used to identify program parts which will likely have a bad performance. In addition, those anti-patterns may contain strategies to circumvent the predicted bad performance. Software performance anti-patterns are a useful instrument to optimize existing software, thus they may be used after the new approach proposed in this thesis reveals that the implemented algorithm or structure does not compete well against other approaches. They are not useful for a general comparison of the performance of data structures and algorithms as they only indirectly influence the differences between several competitors. An implementation which contains an anti-pattern may nonetheless outperform all other competitors.

Brown et al. (2005) [54] determine the instruction and edge coverage of the SPEC JVM98, Java Grande, CaffeineMark and JOlden benchmarks suites. They conclude that those suites "designed to measure real-world applications had poor instruction and edge coverage".

Bergel et al. (2016) [48] choose a set of appropriate benchmarks for every SUT. The benchmarks are performed on every version of the SUT. Doing so, revisions with performance problems can be identified. That is, instead of measuring the performance of a single SUT by the configuration of specific workloads [38], different versions of the same system are compared. Still, this method aims to satisfy specific performance goals as depicted by the chosen benchmarks.

In contrast, Reichelt and Kühne (2016) [161] use the unit tests within the SUT to identify performance changes at the code level. Doing so, also classes of performance problems should be quantified based on the measured data. These approaches on the comparison of system versions give a good overview on the tasks to do, when applying performance tests on a SUT as they also face problems caused by just-in-time compilation, garbage collection and thread scheduling. They are not fully applicable to the problems this thesis is about, as they deal with the comparison of different software versions of the very same system and not with different systems which fulfill a common interface.

Grechanik et al. (2012) [92] develop an automatic performance test generator which uses a feedback-directed black-box system. They cluster the traces in order to keep a maintainable data set and conclude that their approach is effective in the detection of so-called performance bottlenecks, even in large applications. Shen et al. (2015) [181] use a profiler driven by a genetic algorithm in order to identify performance bottlenecks through variating input parameters. Horký et al. (2015) [103] develop a hybrid test generator which automatically generates a performance documentation from the given unit tests. All these approaches, which are mostly derived from the generation of load tests, tend to find performance bottlenecks and misconceptions in a single system. Contrary, the approach developed in this thesis (Chapter 5) is used to compare the performance of different systems with a common interface.

2.6 Performance Testing in Java

As the measurement of a systems computation time is always system dependent, the basics of performance measurement in Java are described here, as well as the work related to the new approach for performance measurement in Java which is described in section 5.4.

Java is listed as one of the most popular and widespread programming languages [61, 21] in use, mostly for its automatic garbage collection and system independence. Along with the abstraction used by the Java virtual machine to provide this independence come several disadvantages regarding a statistically rigorous performance evaluation in Java. This section points out the major threats to Java performance evaluation and methods to circumvent them. In this thesis, the Java HotSpot 64-bit Server virtual machine (*build 25.60-b23*) is used. Although many active Java virtual machine implementations exist [11], only the used implementation will be described in more detail since it is a direct implementation of the Java virtual machine (JVM) specification by Oracle and the underlying HotSpot virtual machine is also used by the widespread OpenJDK project [15].

The main parts of the Java HotSpot Engine [5] which influence the performance of a Java program are the automatic garbage collection, the "ultra-fast" thread synchronization and the just-in-time (JIT) compiler. The first two are of less importance than the latter as the systems designed throughout this thesis are single-threaded and the garbage collection may influence the performance but not the general significance of performance tests if the performance tests are computed often enough and a mean value is used for comparison. In contrast, JIT compilation may alter the structure of the code and optimize such parts which would have caused a relatively worse performance. In more detail, JIT compilation works as follows [5]: For the first times a Java method is accessed, it is interpreted by the *interpreter*. After a certain amount of (user-)time, the HotSpot virtual machine has identified so-called hot spots, i.e. methods which are accessed very often. These methods are then compiled and optimized by the JIT compiler. The most important optimizations defined by the HotSpot Engine specification [5] are:

- Deep inlining and inlining of potential virtual calls. That is, often invoked methods are directly compiled into the calling method instead of invoking them over and over again.
- Fast `instanceof`/checkcast.
- The elimination of range checks if the compiler can prove that the index of the access of an array is within the bounds of that array.
- Loop unrolling decreases the iterations of a loop and increases its body size.
- Feedback-directed optimization. The JIT compiler captures several trace informations of the currently executed code and uses this information to optimistically improve the compiled code. If one of the optimizations violates the assumed behavior of the code, the code is recompiled and re-optimized.

The last three bullet points may alter the code and therefore optimize those parts which are crucial for the performance comparison of a Java program. As described in the IBM Java SDK overview [6], the JIT compiler may reorder, split and delete parts of the control flows. Strictly speaking, the comparison of two Java programs may have different outcomes with and without JIT compilation, as well as between two runs of the same program with JIT compilation. Several recommendations have been made to circumvent the impact of garbage collection, thread scheduling and JIT compilation on performance measurements in Java.

Georges et al. (2007) [87] compare 13 different Java performance evaluation methodologies and their influence on the macroscopic level exemplified by the SPECjvm98 and DaCapo benchmark. The different methodologies are characterized by:

- The number of virtual machine invocations.
- The number of benchmark iterations per virtual machine invocation.
- Activated and deactivated JIT compilation.
- Full compilation before measurement.
- Full-heap garbage collection before measurement.
- Single and multiple hardware platforms.
- The heap sizes and the number of virtual machine implementations.
- Interleaved and back-to-back measurement.

They derive a prototype of a statistically rigorous Java performance methodology and recommend their approach for the future research on Java performance evaluation. Kalibera and Jones (2013) [110] also research on a rigorous Java performance evaluation methodology referring to the SPEC and DaCacapo benchmarks. They concentrate on the detection of a steady state which is the state at which no more optimizations are computed by the JIT compiler and the execution of a performance test can be supposed to be stable. It is argued that the automatic detection of the steady state from [87] is not applicable and that repetition is dependent on the platform, virtual machine and the benchmark. Alghmadi et al. (2016) [30] use performance counters to detect repetitiveness of the measured performance data and decide when to stop performance tests. All approaches have in common that they target macroscopic JVM benchmarks and try to find the ideal set up for benchmark runs and JVM configuration to produce feasible and stable performance results. Contrary, in this thesis, a set of method calls for each system under test should be generated in order to perform tests for a better comparability.

On the microscopic level, Rodriguez-Cancio et al. (2016) [163] isolate the segment under analysis (SUA), e.g. a loop, via slicing and produce payloads that reflect the input of the application under test (AUT) on the SUA. The main goal is to prevent dead code elimination and constant folding during a micro benchmark by the JVM. Thus they investigate the same threat to performance tests as investigated in this thesis. In contrast, the performance comparison in this thesis is used to analyze the complete SUT and not only parts of it. In addition, neglecting automatic optimizations would not stand the requirement of fair performance comparison. Implementations which rely on the assumed code optimizations by JIT compilation would be disadvantaged. Nonetheless, the general approach on a rigorous Java performance methodology and the awareness of the distortive influence of JIT compilation is used to perform rigorous performance tests in this thesis. The general approach is described in Section 5.4.

2.7 Software Performance Engineering

In this section, the approaches on benchmarking, workload generation and performance measurement proposed in this thesis are set in relation to the general field of Software Performance Engineering.

Definition 2.3 *Software Performance Engineering (SPE) represents the entire collection of software engineering activities and related analyses used throughout the software development cycle, which are directed to meeting performance requirements. [207]*

Along the given definition, [207] divide SPE into two general approaches: First, the model-based approach, which uses performance models early in the development cycle and incorporates certain performance expectations into the system requirements. Secondly, the measurement-based approach which is applied late in the development cycle when the SUT can be run and the performance can be measured. This thesis deals with already implemented structures such that a measurement-based approach is used. Other definitions of SPE [183] do not accept the measurement-based approach and solely focus on the model-based approach. As the complete software development cycle is ignored and only the concrete implementation is necessary for this thesis, any further discussion on SPE is spared. The general approach on performance test comparison proposed in this thesis may be used as the final stage in a SPE approach but it is not a SPE framework in itself.

2.8 Software Testing

As this thesis provides a new approach for the automated generation of software performance tests and benchmarks based on software test generation, the basics on software testing in general are presented in this section.

The field of Software Engineering distinguishes several techniques to analyze and evaluate the quality of software through the process. Those techniques might be divided through the scheme depicted in Figure 2.9 [122].

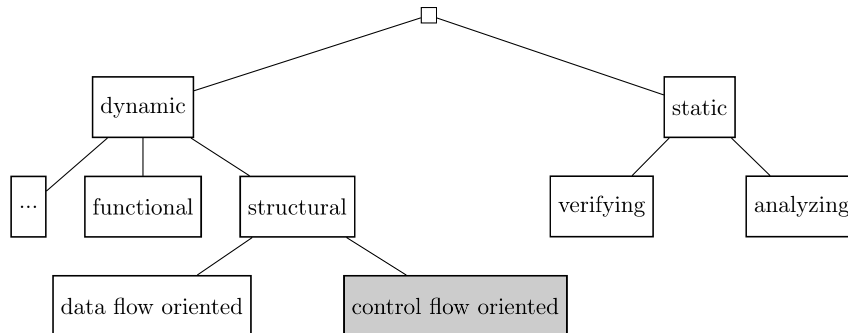


Figure 2.9: Overview of software quality checks (translated from [122])

It only shows a part of the classification as this is sufficient to characterize the software tests used in this thesis. Static techniques do not execute the system under test (SUT), do not produce test cases and may not be used to make thorough statements about correctness and soundness. Instead, they analyze the SUT using static criteria or verify systems by mathematical models. Dynamic techniques execute the SUT with certain input values. They may not be used to make thorough statements as well and are more of a sample. Besides diversifying and other approaches, the dynamic techniques may be divided into functional and structural approaches. Functional approaches test the correct functional behavior of the SUT, without covering the structure of the system itself. Structural approaches use insights on the

structure of the SUT (see Figure 2.10). Data flow oriented approaches try to cover all memory reading and writing accesses in the SUT. Control flow oriented techniques try to maximize the coverage on the control flow of the SUT. That is, a system that is executed with a control flow oriented technique should access a maximum of a predefined part of the control flow, e.g. all statements.

For dynamic functional approaches a lot more input of the developer is required as for dynamic structural approaches. The developer needs to interpret the specification and test cases are then generated from the developers interpretation of the specification (see Figure 2.10). The requirement of an automated and by all means fair comparison of several structures forbids the often subjective input of a (human) software tester. Instead, an automatic dynamic structure based approach may be able to automatically analyze the SUTs and automatically create suitable test cases afterwards.

Hoffmann (2013) [100] defines four layers in software testing as depicted in Figure 2.11. The code or module level at the bottom is abstracted by the integration level where several modules are combined with each other. A test on the module level would only test on the correctness of the module itself whereas the integration tests test the interaction between several modules. On the system level, the complete software system is tested and this is often considered to be the last test before the deployment of software to the customer. The customer may perform an acceptance test and then accept or reject the completed software.

The techniques described in this thesis operate on the module or unit test level. Every structure compared to other structures is considered to be its own module. Those modules could be integrated into a larger system and for the purpose of performance tests it also may be beneficial to measure performance on the integration level. Index structures for instance need to be integrated into a database and maybe connected to a structured query language (SQL) system. But as for the tests of correctness the behavior on the lowest level has to be determined before the overall performance can be analyzed. The current research on automatic unit test generation is described in section 2.11.

2.9 Control Flow Terminology and Coverage

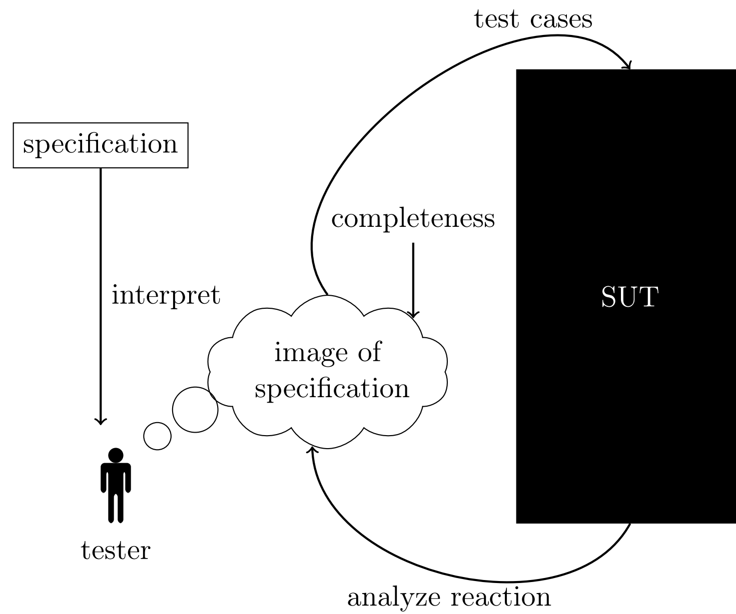
This thesis uses the coverage of test sets on the control flow of a system as optimization goal for a new approach on automated performance test generation. The basics on control flows and their coverage are presented in this section.

The structure oriented test approach is based on insight knowledge of the SUT, i.e. on its control flow. One way to describe the control flow of a program is its control flow graph (CFG) which is described in this section. The definitions in this section are mostly derived from [98].

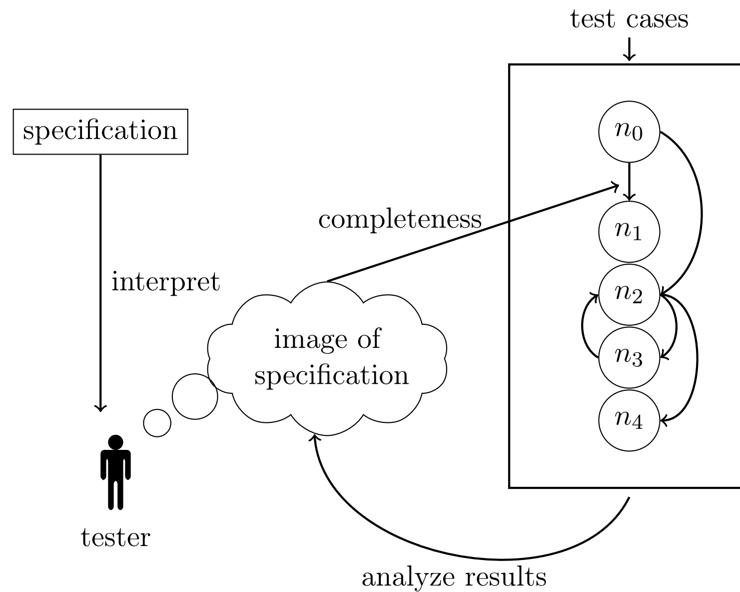
A *flow graph* $G = (N, E, n_0)$ is a directed graph containing a set of nodes N, a set of edges $E \subseteq N \times N$ and a start node $n_0 \in N$.

A *control flow graph* (CFG) is a flow graph that represents the control flow of an imperative program P. Each node in a CFG represents a single statement of P. Often a sequence of statements that are always executed together is combined into a single *basic block* and a node then represents one basic block. An edge (u, v) between two nodes u and v indicates a jump from u to v. In McMinn (2004) [128], edges which depict conditional jumps are called branches.

Figure 2.12 shows the CFG (middle) of a program (left). Every node represents a *basic block*. For instance, the nodes n_0 and n_3 contain several statements that are always executed



(a) Function oriented test



(b) Structure oriented test

Figure 2.10: Test classification (translated from [122]). Using function oriented tests (a), the tester creates test cases following his image of the specification and analyzes the results with respect to the specification. Structure oriented tests are built using insights on the SUT, the test results are then analyzed with respect to the specification. For both approaches, the tester has to interpret the specification.

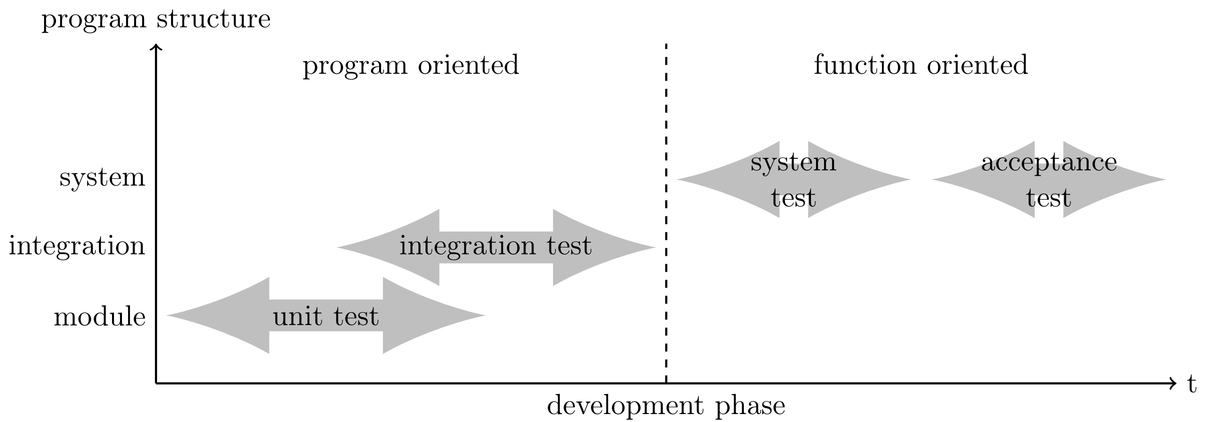
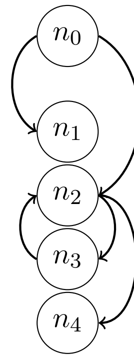


Figure 2.11: Four layers of software testing [100]

```
int iterative(int n){
  int f = 1;
  if (n < 0)
    throw new Exception();
  while(n > 0)
    f = f*n; n = n-1;
  return f;
}
```

(a) Java method, returns the factorial of n.



(b) Corresponding control flow graph (CFG). Every basic block is one node in the CFG.

$DOM(n_0) = \{n_0\}$

$DOM(n_1) = \{n_0, n_1\}$

$DOM(n_2) = \{n_0, n_2\}$

$DOM(n_3) = \{n_0, n_2, n_3\}$

$DOM(n_4) = \{n_0, n_2, n_4\}$

(c) Dominance relationships.

Figure 2.12: Program code, CFG and dominance relationships

together. If a program is executed, the sequence of nodes corresponding to the executed blocks is called a *path*. For instance, the execution of the *iterative* method with the actual parameter $n = 1$ results in the path $p_1 = (n_0, n_2, n_3, n_2, n_4)$ and $n = 2$ results in the path $p_2 = (n_0, n_2, n_3, n_2, n_3, n_2, n_4)$.

Considering p_1, every basic block except for n_1 has been executed at least once. That results in a *basic block coverage* of 0.8 following the formula

$$C_0 = \frac{|basic\ blocks\ at\ least\ executed\ once|}{|basic\ blocks\ in\ CFG|}. \tag{2.3}$$

In the literature, the basic block coverage is also denoted as statement coverage [122]. Some statements in the source code may be split into several statements in the object or machine code. Using the statements derived from the source code for the calculation of the statement

coverage may also produce contorted results. In this thesis, the term of basic block coverage is used and it should be mentioned that this is denoted as statement coverage in most of the used literature.

Path p_1 contains at least one execution of four of the five branches, resulting in a *branch coverage* of 0.8 following the formula [122]

$$C_1 = \frac{number\ of\ branches\ at\ least\ executed\ once}{total\ number\ of\ branches\ in\ CFG}. \tag{2.4}$$

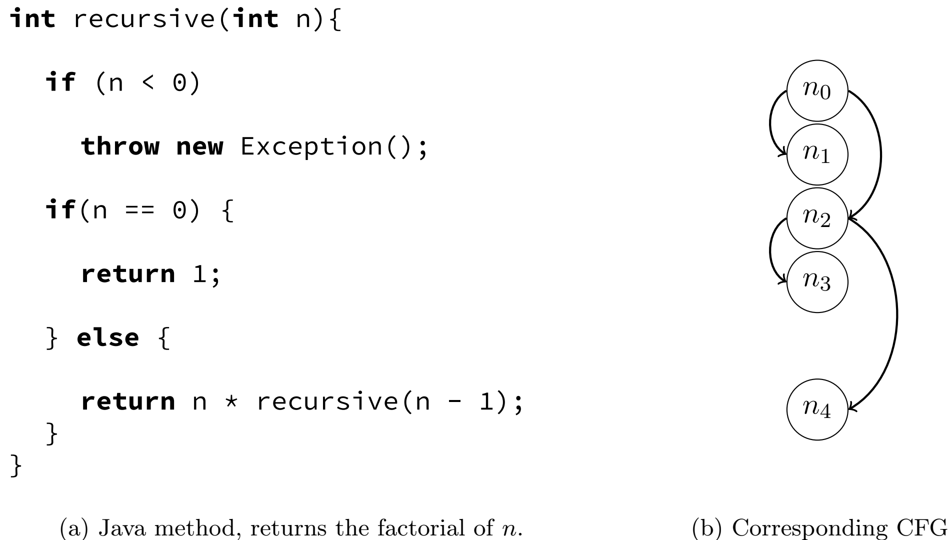
The *condition coverage* is used to describe the executed branch conditions of a CFG. The CFG in 2.12 contains two branch conditions, one in n_0 and one in n_2. The simple condition coverage requires the execution of every condition at least one time with the result `true` and at least one time with the result `false`. To achieve a full *simple condition coverage* for n_0 it must be executed with $n < 0$ at least once and also with $n >= 0$ at least once. The *condition/decision* coverage additionally requires that the combined condition is at least executed once with `true` as well as `false` as result. For instance, the decision `a && b` contains the conditions `a` and `b` which both should be evaluated with `true` and `false` at least once and also, `a && b` itself should be evaluated with `true` and `false` at least once. The *minimal multiple condition coverage* also requires that all combinations of conditions inside each decision must be evaluated [122].

Although the paths p_1 and p_2 share the same basic block, branch and condition coverage, they have to be respected as different. The *complete path coverage* is not applicable if the CFG contains an infinite loop. The *boundary-interior-test* contains a boundary and an interior test. The boundary test requires one execution of each loop and the execution of each path in that loop. The interior test requires that all paths which can be executed within two iterations of a loop are executed [122].

In a CFG, a loop is defined through a *header* and a *back edge*. The header is a node in the CFG and every node in the loop can only be accessed through the loop's header. The back edge is an edge from one of the nodes in the loop to the loop's header. For instance, n_2 is a header and the edge (n_3, n_2) is the corresponding back edge of the while loop. To find possible loop headers, one has to compute the dominance relationships in a CFG. A node x dominates a node y in a CFG if every path in the CFG from its initial node to y contains x. It is also defined that $DOM(y) = \{x | x\ dominates\ y\}$. The only dominator of the start node is the start node itself ($DOM(n_0) = \{n_0\}$). The dominators of any other node is the union of the sets of dominators for all predecessors of the node. The node itself is also in that set of dominators ($DOM(n_i) = (\cup_{p \in preds(n_i)} DOM(p)) \cup \{n_i\}$) [98]. A node n_i in a CFG g is a loop header, if a node $m \in g$ exists with $n_i \in DOM(m)$ and $(m, n_i) \in g$. A loop is identified by its back edge. Every execution of a loop header denotes one entry in the corresponding loop(s) and the execution of one back edge denotes the iteration of the loop.

Figure 2.13 shows the same algorithm as 2.12 but implemented recursively. Thus, the corresponding CFG does not contain a loop, although the recursive implementation indirectly performs the same number of iterations as the iterative implementation. Tracking recursive executions or "loops" over several method invocations lies beyond the capabilities of a CFG.

One test case may often not be sufficient to achieve a full coverage. Therefore, several test cases are combined into one *test set*. The coverage is then measured after all test cases in one set have been executed.



(a) Java method, returns the factorial of n.

(b) Corresponding CFG

Figure 2.13: Recursive code fragment and CFG

2.10 Code Coverage in Java

Some specialities on code coverage arise through the structure of the generated byte code [124]. This section displays those specialities, the approaches to measure different coverage types in Java and some of the most recent implementations.

Figure 2.14 shows a simple Java code snippet, its CFG, the corresponding byte code and the CFG resulting from the byte code. Basically, for any decision, all combined conditions are split up into atomic decisions. That does not only result in a much larger CFG compared to the CFG resulting from the source code. Using only atomic decisions in the CFG causes that a CFG, which has full branch coverage, also has full simple condition coverage.

The possibilities of implementing code coverage in Java are listed in Figure 2.15. Runtime Profiling approaches like the Java Virtual Machine Profiler Interface (JVMPI) [12] or the Java Virtual Machine Tool interface [13] for newer Java versions dynamically measure executing programs. Profiling often has a high impact on the overall performance of a program and is primarily used for performance analysis. Through instrumentation, code snippets like flags or counters are inserted at specific positions in the source or byte code. Every time a program is executed, the instrumented code, or probe, is executed as well when the corresponding code block is executed. This probe may then be used to count the number of accesses on a code block. For instance, in Figure 2.14, a probe may be inserted above the lines 0 and 5 to reflect the execution of the nodes n_0 and n_1, respectively. The execution of a probe indicates the execution of the following byte code instructions until the next probe, e.g. branching node. Such a probe may contain the increment of a static variable associated with the corresponding CFG node. After execution that variable shows how many times the CFG node has been accessed. The complexity and position of the used probes determine the possible coverage types and the influence on the overall performance of the original program. Instrumenting the source code usually is more costly than byte code instrumentation in terms of runtime. Without the knowledge of the source code, it is not possible to evaluate decisions which consist

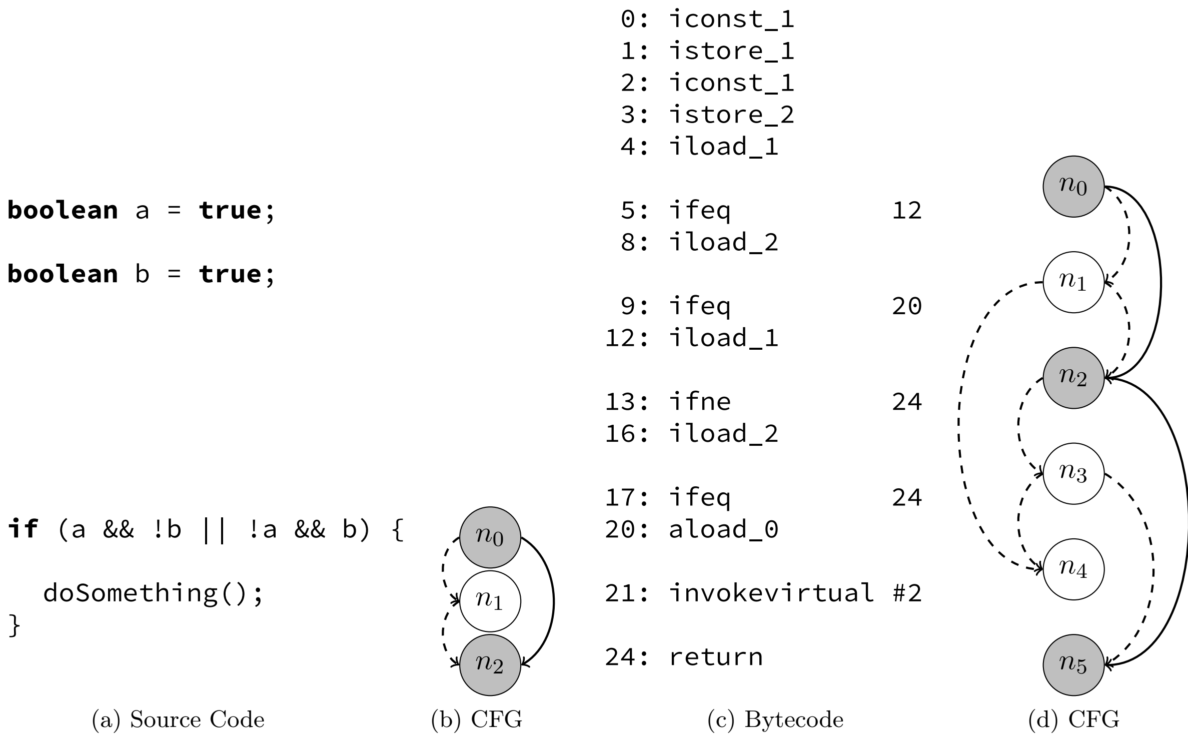


Figure 2.14: Control Flow Graph generation on the Java source and byte code generation. The gray CFG nodes represent nodes which have been accessed when the program is executed. The dashed branches indicate that these branches have not been traversed during execution.

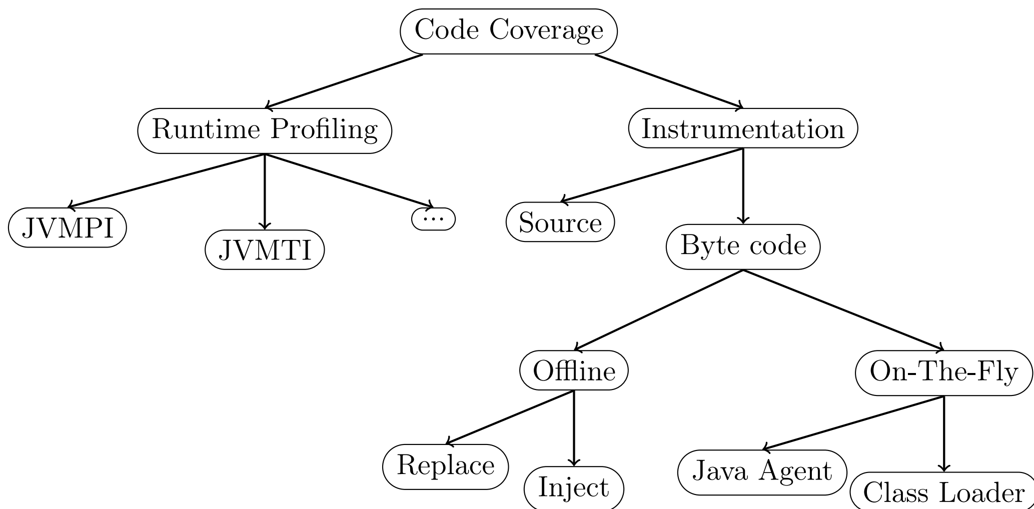


Figure 2.15: Hierarchy of code coverage implementation techniques in Java [9]. JVMPI and JVMTI are tools for runtime profiling, the other leaves are general techniques.

of several connected conditions. Byte code instrumentation can be performed on several levels. Offline instrumentation changes the already compiled class file either by replacement or injection. Using replacement, the original class files are not altered, but new class files are generated which contain the additional probes. Contrary, by using injection, the class files are changed and the additional probes are directly inserted into the existing files. Injection may be performed dynamically or *on the fly*. That is, the byte code instrumentation is performed during the runtime of the program. Using a class loader, the Java byte code is instrumented whenever a desired class is loaded by the virtual machine. Every class that is loaded by the virtual machine may be manipulated by a Java Agent. Overriding the class loader in Java is more modular, but the Java Agent can be used on any class, loaded by any class loader.

Table 2.3 lists a subset of currently available Java coverage tools. In addition, a proven correct extraction of the CFG from Java byte code can be found in [33]. For Table 2.3, only those tools are chosen which at least provide informations about the used approach and the supported coverage types. This excludes most proprietary coverage tools. Three of nine tools use source code instead of byte code instrumentation. The use of source code instrumentation may result in another sort of statement coverage than using byte code instrumentation. Source code statements are often split up into several byte code instructions. Furthermore, all of the described tools do not state clearly if actually basic blocks instead of single statements are covered. For both cases, no other statements than the actual executed ones are covered but they are summed up differently and the coverage result may be distorted between the statement coverage measured by two different tools. For instance, in Figure 2.14d node n_0 represents two statements, six byte code instructions and one basic block. It has to be noted that it only represents one quarter of the decision evaluated by the if-statement. The nodes n_1,n_2 and n_3 represent another quarter of the statements but two byte code instructions and one basic block. Node n_4 represents no statement, one basic block and two byte code instructions. At last, n_5 represents one statement, one basic block and one byte code instruction. As displayed in Figure 2.14, the execution of the code snippets causes a statement coverage of $\frac{3}{4}$, a basic block coverage of $\frac{3}{6}$ and a coverage on the byte code instructions of $\frac{9}{15}$. By some of the tools, statement coverage is described as *line* coverage (CLOVER [2]) or the coverage on byte code instructions as statement coverage (ECLEMMA [101]).

2.11 Automatic Test Generation

This thesis uses the current achievements in automated test generation to automatically generate performance tests. This section not only outlines the current achievements in unit test generation but also in GUI (graphical user interface) test generation. GUI tests are created on the basis of an interface which is defined by all accessible components on the GUI itself. The research in GUI tests shall provide insights on how the generation of interface-based test cases can be achieved. In addition, the term of *sequence based* test generation is often used for the same purpose as the interface based test generation in this thesis. That is, sequence based test generation assumes that for most systems, only the invocation of a sequence of methods will change the state of a system under test (SUT). For the remainder of this thesis, a test case always contains a sequence of method invocations and this sequence may have a length of one. The current state of performance test generation, which is used to find bottlenecks in software programs or perform load tests, is presented in Section 2.5.

Table 2.3: Summary over current code coverage tools for Java.

Name	Coverage-Types	Approach	Version	Source	License	Notes
Clover	line, branch, method	source code instrumentation	18.04.2017	[2]	Open Source	
Cobertura	statement	dynamic byte code instrumentation	2.1.1. (01.06.2016)	[65]	GNU GPL 2	only supports up to Java 7
CodeCover	line, branch, loop, MC/DC	source code instrumentation	21.09.2014	[176]	EPL	supports Java and COBOL
Coverlipse	statement	dynamic byte code instrumentation	02.04.2013	[111]	CPL	only supports up to Java 1.4
EclEmma (JaCoCo)	statement, branch, method, type, complexity	dynamic byte code instrumentation	3.0.0 (28.06.2017)	[101]	EPL	Eclipse Plugin based on JaCoCo
JCov	statement, branch, method	static and dynamic byte code instrumentation	262 (27.08.2017)	[125]	GNU GPL 2	supports Java 9
JCover	line, branch	source and/or byte code instrumentation	12.11.2009	[8]	Proprietary	only supports up to Java 1.4
Serenity	statement, complexity	dynamic byte code instrumentation	03.09.2016	[69]	Open Source	Jenkins-Plugin
Testwell CTC	statement, method, decision, condition, MC/DC	source code instrumentation	8.2 (16.05.2017)	[20]	Proprietary	Plugins for various IDEs

For most of the presented techniques it should be noted that a test is always meant to be connected to a corresponding test oracle which defines whether the outcome of a test case is correct or not. In a recent survey, Barr et al. (2015) [40] state that the generation of oracles is still an open problem and "Much work on test oracles remains to be done". As the general correctness of a test is irrelevant for a performance test, which should prove all possible input sequences, the further discussion on test oracles is spared.

In terms of this thesis, test generation is always meant to be automatic but not generated by a user defined set of parameters. That is, automatic test generation shall derive all pieces of information needed for the generation of test cases from the system under test (SUT) itself. For instance, [16] generates test data by using a model which has to be defined by the user. The automation is then only limited to the repetitive creation of test cases on the basis of the model with random variations in a predefined parameter space. Therefore, this type of test generation is classified as workload generation, a part of benchmark generation which is discussed in 2.3.

The current achievements in automated test generation may be divided in four major approaches: Random Testing, Model Based Testing, Concolic Testing and Search Based Testing. Each approach is described in more detail in the following subsections. Each of the described approaches is usually implemented by a test generation tool. Both, the name of the approach and the name of the tool, are used interchangeably in the literature and in this thesis.

2.11.1 Random Testing

Random test generators randomly select possible input parameters for the SUT or parts of it. The main problem of randomly generating test cases is that these test cases are often not valid in terms of the actual specifications of the tested system. Two types of invalidity have to be strictly divided. On the one hand, certain inputs cause errors predicted by the program and are somehow handled, e.g. with an exception. On the other hand, invalid inputs which are not contained in the programs preconditions are called *contract-violating* tests. JCrasher [70] executes test cases in order to produce exceptions and errors and tries to distinguish between contract violating and invalid inputs. As Java does not provide method preconditions by itself, those are derived from the type of the exception and its message itself. Jartege [145] requires the definition of Java classes in the Java Modeling Language (JML) to define the preconditions. Within the parameter space of those conditions, Jartege then randomly creates test cases. RANDOOP [147, 146] creates test cases incrementally. A method sequence is expanded by adding a random method invocation with random parameter values from previously constructed sequences. Sequences that cause illegal behavior, like an unexpected exception, are discarded. But sequences which violate a contract are added to the output list of *contract-violating* tests. In addition to some predefined contracts like reflexivity and equality in the Java Collection Framework, the user may define a *contract-checking interface* to extend the system with custom contracts. Also, methods may be annotated to be ignored by RANDOOP representing invariants or to create regression assertions. Finally, those sequences which cause normal, valid behavior are added to the list of *regression tests* (see Figure 2.16). T3 [158], which is a complete and enhanced recreation of T2 [160], creates random test sets just as RANDOOP and directly supports the Java stream API and closures. Those may also be used with T3 to unfold faulty interactions between methods. Prasetya (2013) [158] claims that T3 outperforms RANDOOP and is outperformed by EVOSUITE [81]. They argue that this is caused by the fact that T3 does not use instrumentation due to the generally random

approach. T3 is the back end of iT3 [159], a Java testing tool which not only allows the automatic generation of test sets but replaying and querying test suites in Java. GRT [126] guides the generation of random tests using five modules: (1) The constants of a program and the likelihood of their usage are extracted by a static mining operation. (2) Impure methods, thus methods which have a side effect on the states of the classes objects, are extracted and used to change the object state. (3) All created method sequences are stored in an object pool together for the generated types for reuse. (4) Those demands are dynamically satisfied for methods which cannot be accessed without the usage of external libraries. The resulting sequences and objects are stored in a separate object pool. (5) Finally, the guidance of random generation is achieved by weighting the methods which are being invoked in the tests. They are weighted by incorporating the number of branches in a method that are uncovered, the ratio of successful and failing invocations and the number of times the method has been selected since the last update. The updates are not performed constantly but after a certain time, in order to minimize the cost of the generation. JTExpert [170] uses a guided random search approach. The initial instances are used as seeds for the following search for suitable test sequences based on those seeds. When invoking randomly selected methods, the seed instance generator is used to add the desired parameters. After static analysis, the test generator randomly selects a branch b from all targets to be covered T and removes b from T. Then a random method sequence is generated and executed. All branches covered by the executed sequence are removed from the branches to be covered. This may not include b. If b has not been covered yet, it is added to a set of branches which is computed in the next run of the algorithm, after T is empty. The algorithm stops if no more branches are to be covered or a certain time limit is reached.

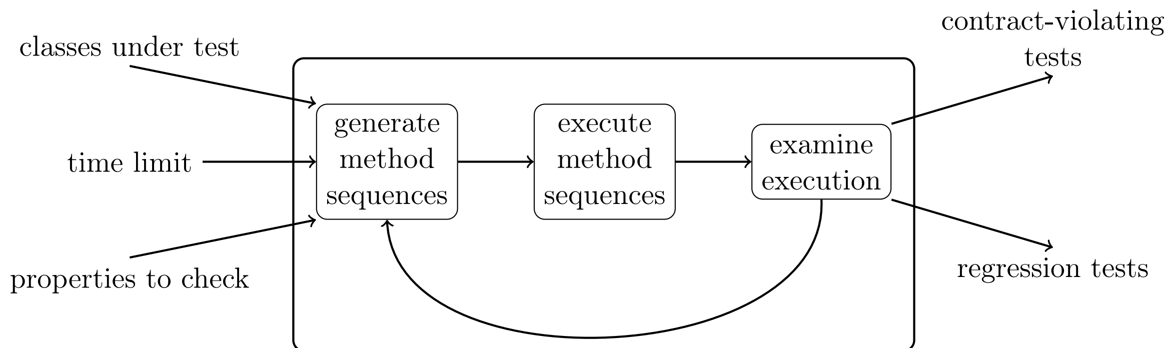
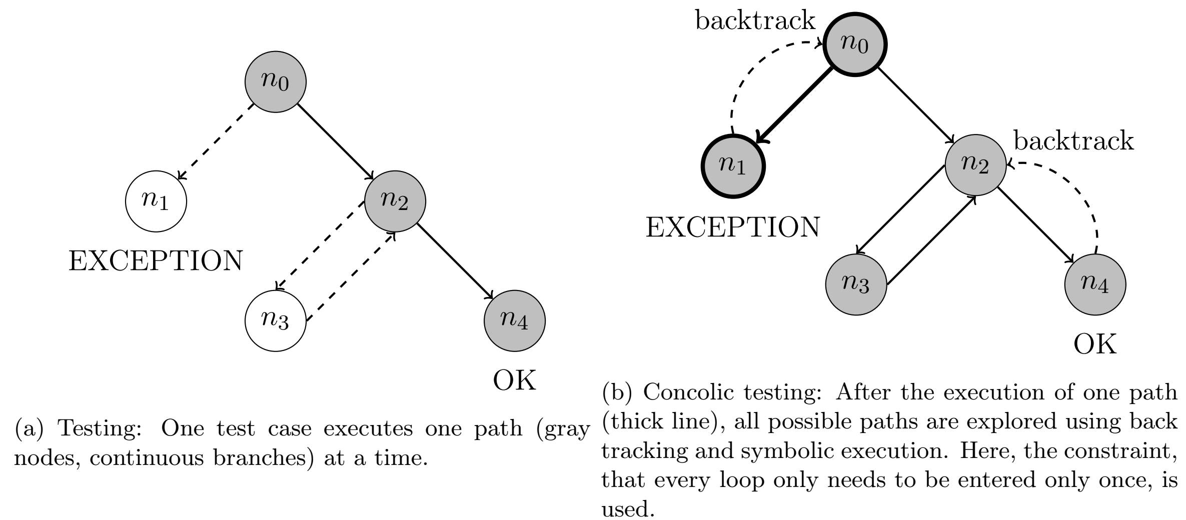


Figure 2.16: Behavior of RANDOOP [146].

2.11.2 Concolic Testing

Symbolic execution uses symbolic values instead of actual values when executing a program. The symbolic values are incorporated into a path condition as the execution emerges. The path condition denotes which values the several input parameters are required to have in order to reach the current part of the program. After the execution of one concrete path, a symbolic execution framework may use backtracking to generate all possible paths in a control flow graph (CFG). This is called *concolic* testing [88]. Figure 2.17 exemplifies concolic testing.

After one random path is executed, all remaining paths are evaluated using backtracking. Examples for symbolic execution frameworks are Java PathFinder (JPF) [151, 150, 153] for Java or Euclide [89] for C programs. Symbolic execution shall be used for critical software systems which require a high confidence on the unit tests or in bug finding, for example in NASA software [152]. Symbolic execution is also applied in model checkers which try to find a failing or illegal path. All model checking approaches may be applied on test generation.



(a) Testing: One test case executes one path (gray nodes, continuous branches) at a time.

(b) Concolic testing: After the execution of one path (thick line), all possible paths are explored using back tracking and symbolic execution. Here, the constraint, that every loop only needs to be entered only once, is used.

Figure 2.17: Testing and Concolic testing examplified on the iterative factorial method (see Figure 2.12)

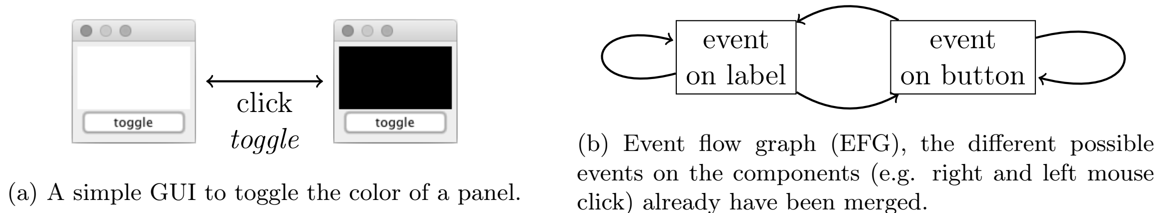
PathCrawler [206] is a prototype for the automatic creation of test cases which fulfill the all-paths criterion, i.e. test cases that cover all possible paths of a program using symbolic execution. To avoid path explosion caused by loops, the number of required loop iterations can be user limited by the constant k. DART (Directed Automated Random Testing) [88] works in three steps. First, using static source code parsing, an interface of the SUT is automatically extracted. Secondly, due to random test generation on this interface, the most general parameter space is explored. Thirdly, new tests are systematically generated along the paths generated in the second step. DART is one of the first tools that uses concolic testing. Sen et al. (2005) [178] combine concolic and concrete program execution to generate unit tests in their Concolic Unit Testing Engine (CUTE). There, the focus lies on the handling of pointers in C programs. In the tool jCute [177] this approach is transferred to the Java programming language. EXE [58] uses symbolic execution to generate failing test cases. Based upon EXE, KLEE [57] creates test sets with high coverage. The tools Barad [86] and ACTIVe [34] use symbolic execution on the event handlers of Java applications to create GUI tests with a high coverage. Jensen et al. (2013) [109] combine concolic testing and an event handler model to create test cases for Android apps. A more performance related approach is WISE [56] which creates worst-case inputs for a SUT. Although symbolic execution is a robust test generation technique, it is very costly in terms of generation time and required hardware resources.

By using symbolic execution, all possible paths of a SUT are discovered. A loop without a proper ending condition may always provoke a very high yet infinite number of paths which

is called path explosion. Therefore, the usage of symbolic execution may not be applicable for the SUTs investigated in this thesis.

2.11.3 Model Based Testing

Just like the JML definitions in Jartege [145], model based approaches rely on an abstracted definition of the SUT. The UML modeling language is often used as basis for such definitions. For instance, UML Use Case and Activity diagrams [197, 63] or statechart diagrams [171] are used. In terms of GUI testing, the event flow graph (EFG) may be used [132] as a suitable model. An EFG consists of nodes, each representing an event on a GUI component, and edges, one for each possible action which leads from one event on a GUI component, i.e. node in the EFG, to another. Figure 2.18 displays a simple GUI and the corresponding EFG. Menninghaus et al. (2017) [135] compare local search algorithms and genetic algorithms for automated GUI test generation. There, the coverage on the SUT or the coverage on the corresponding CFG are used as optimization goals. The analogy of an EFG in the terms of interface based test generation may be a graph which contains method invocations as nodes and each possible following method invocation as an edge to another node, i.e. invocation. [210] additionally define the *event semantic interaction* (ESI) relationship between two events and the corresponding graph, the ESIG. It is traversed in order to produce sequences of GUI interactions which are likely to change the runtime state of the program.



(a) A simple GUI to toggle the color of a panel.

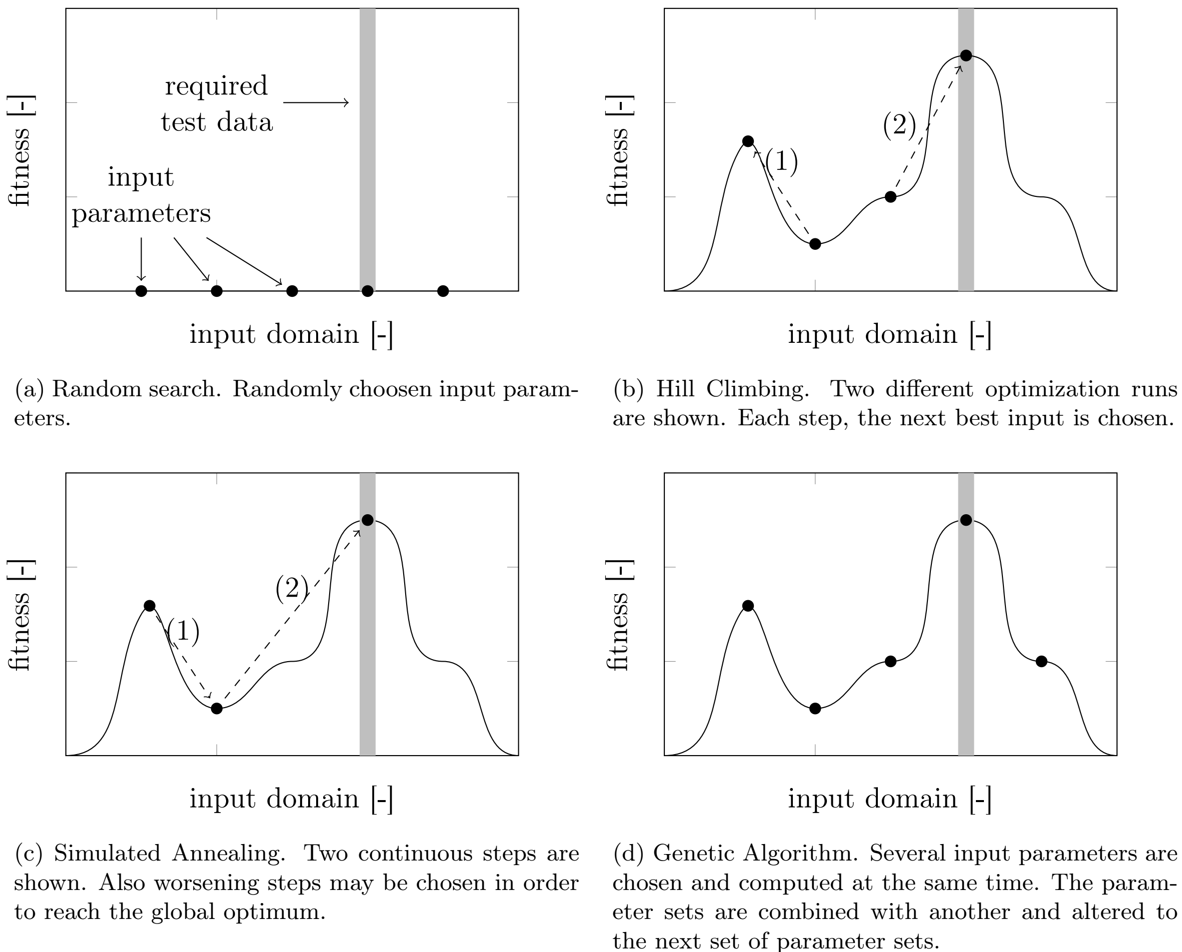
(b) Event flow graph (EFG), the different possible events on the components (e.g. right and left mouse click) already have been merged.

Figure 2.18: A simple GUI and the corresponding event flow graph (EFG).

2.11.4 Search Based Testing

Search based software testing (SBST) [129] uses meta heuristic search algorithms such as simulated annealing, hill climbing and genetic algorithms to systematically explore the search space. Figure 2.19 visualizes the key idea of search based software testing. Given an input domain, for instance all possible input parameters of a method and a fitness function, which depicts the quality of the desired outcome and the neighboring solutions, a fitness landscape is defined. In terms of structural testing, the fitness function would usually depict the code coverage on the method when executed with the given input parameters. The input domain may not only contain the possible input parameters for one method but for all accessible methods and initializers in the SUT. Search algorithms explore this fitness landscape and try to reach the global maximum of the fitness function. In the ideal case, the global maximum of the fitness function equals the part of the input domain which is required to create the desired test set. Random search would randomly choose input parameters until either a resource limit, e.g. time, is reached or the required input has been chosen. Random search does not

use any guidance and does not depend on a fitness function. Local search algorithms like hill climbing and simulated annealing [168] try to enhance the search stepwise. In each step one of the neighboring input parameters is chosen regarding to the current state of the approach. In hill climbing, the next state is simply a neighboring state with a better fitness, e.g. with the maximum fitness. It stops, when no new maximum can be reached from the current state and therefore may get stuck in a local optimum. In contrast to hill climbing, simulated annealing uses an additional temperature function. It indicates how far the search reaches out for an optimum. The longer the algorithm runs, the colder the temperature gets. The colder the temperature is, the more unlikely it is for the algorithm to choose another state if the current state is a local optimum. A step may not only contain the selection of new input parameters but another method or another constructor when the SUT contains several classes and methods.



(a) Random search. Randomly choosen input parameters.

(b) Hill Climbing. Two different optimization runs are shown. Each step, the next best input is chosen.

(c) Simulated Annealing. Two continuous steps are shown. Also worsening steps may be chosen in order to reach the global optimum.

(d) Genetic Algorithm. Several input parameters are chosen and computed at the same time. The parameter sets are combined with another and altered to the next set of parameter sets.

Figure 2.19: Four approaches on search based software testing.

2.11.4.1 Genetic Algorithms

A genetic algorithm [102] starts with an initial, e.g. random, population of solutions called generation. The next generation is created by selecting a set of solutions from the previous generation with respect to the fitness function. The solutions of every generation are then recombined (e.g. crossover) and also mutated. Recombination usually means that parts of a solution, e.g. the statements in a test case, are rearranged inside a single solution or exchanged through several solutions. Mutation alters a single solution, e.g. altering the input parameters of a method invocation in a test case or adding new statements to a test case. Thus, several test sets are checked at once and ideally, the best parts of one generation are combined to create an even better generation of test sets. The general procedure of a genetic algorithm is depicted in Algorithm 1. Note that all parts of a genetic algorithm are customizable. The evaluation depends on the chosen fitness function. Selection, mutation and recombination are all independent and exchangeable modules which may have an impact on the change rate of the algorithm. Also, genetic algorithms rely on a widespread initial population in the search space. The impact of different seeding strategies in search-based unit test generation is investigated in Rojas et al. (2016) [164].

Input : Stopping condition C, Fitness function δ, Population size p_s, Selection function s_f, Recombination function r_f, Recombination probability r_p, Mutation function m_f, Mutation probability m_p
Output: Population of optimized individuals P

```
1  t ← 0;
2  P_0 ← GenerateInitialPopulation(p_s);
3  PerformFitnessEvaluation(δ,P_0);
4  while ¬C do
5  |   P_{t+1} ← {} ; // next generation
6  |   while |P_{t+1}| < p_s do
7  |   |   p_1, p_2 ←Selection(s_f,P_t);
8  |   |   o_1, o_2 ←Recombination(r_f,r_p,p_1,p_2);
9  |   |   Mutation(m_f,m_p,o_1);
10 |   |   Mutation(m_f,m_p,o_2);
11 |   |   P_{t+1} ← P_{t+1} ⋃{o_1, o_2};
12 |   end
13 |   t ← t + 1;
14 |   PerformFitnessEvaluation(δ,P_t);
15 end
```

Algorithm 1: Pseudo-code for a simple genetic algorithm. Here, the recombination is a crossover between two solutions.

Four of the most common mutation and recombination operators are described in the following paragraphs. Their usage in this thesis is described in Section 5.3.1.

HUX The half-uniform crossover (*HUX* [96]) operator swaps half of the bits that are not equal in two given integer variables with one another.

SBX Deb and Agrawal (1994) [73] propose the simulated binary crossover (*SBX*) for the variation of real, i.e. floating point values in a genetic algorithm. It is designed with respect to a single-point crossover of binary-coded variables and uses two of its properties: the average property and the spread factor property. The average property denotes that the average of both variables incorporated in the crossover is the same before and after the crossover is applied. The spread factor property denotes that a spread factor of $\beta \approx 1$ is more likely than any other β value, where β is defined as the ratio of the spread of the offspring values c_1, c_2 to that of the parent points p_1, p_2 with $\beta = |\frac{c_1 - c_2}{p_1 - p_2}|$. The distribution used by the *SBX* to generate offsprings is defined as $0.5(n+1)\frac{1}{\beta^{n+2}}$, where n is the distribution index. Larger values of n let the generated offsprings be closer to their parents.

BitFlip The *BitFlip* operator goes through each bit of an integer variable and flips it if a user defined probability is met. In detail, a `0` becomes a `1` and vice versa.

PM The polynomial mutation (*PM*) [74] is used to simulate a bit flip operation on a real, i.e. floating point value. Like the *SBX*, it uses a distribution index which indicates how close the offspring values are located to the parent values. In more detail, the offspring c is created by using a perturbance factor $\delta = \frac{c-p}{\Delta_{\max}}$ with the parent value p and maximal permissible perturbance $\Delta_{\max}$. The mutated value is calculated as $c = p\delta\Delta_{\max}$.

Besides this general usage of genetic algorithms, automated search based testing frameworks use search based software testing to automatically create structure based tests, i.e. tests with a high coverage [129]. TestFul [39] uses a hybridization of class and method based fitness along with the combination of genetic algorithms and hill climbing. In the outer loop, a classic genetic algorithm is used to generate test sets with a high structural coverage on the SUT. The test sets contain constructor calls, method invocations and assignments of primitive values. Instead of working only on the class level of the SUT, the inner loop solely considers a single method. In the mutation phase of the outer loop, the parameters of single methods are chosen by targeting the conditions in the method. A branch is connected to one condition on the byte code level. It is divided between reachable and not reachable branches together with evaluated and not evaluated conditions. The distance of a test to a desired branch is computed with respect to the condition that forms the branch on the byte code level with the following formula:

$$distance(a \oplus b) = \begin{cases} +\infty & \text{condition not evaluated} \\ -\infty & \text{target branch executed} \\ |a-b| & \text{otherwise} \end{cases} \tag{2.5}$$

The variables a and b are the two parts that form a condition which may either be constants, local variables or field variables. For instance, if the condition is $(a == 1000)$, a would be set to the current value when the condition is evaluated and b to 1000. Baresi et al. (2010) [39] argue, that $a \oplus b$, where $\oplus$ is any relational operator, are the only conditions found in the byte code. Using the distance, hill climbing is used to enhance the coverage of a single method invocation before going back to the outer loop. In the outer loop, the recombination and selection ensures that method invocations with parameters that cause a high coverage are spread out to the entire population. Baresi et al. (2010) [39] claim that TestFul outperforms the tools RANDOOP [177], jAutoTest [118, 7] and ETOC [189]. Therefore, those tools will not be explained any further.

The tool EVOSUITE [81, 82] also uses a genetic algorithm to automatically create test sets with a high structural coverage. The fitness function is computed using two main approaches of search based software testing: the *branch distance* and *whole test suite* (WS) generation. With branch distance, the one introduced by Korel (1990) [115] is meant. As the branch distance is of major interest for every recent approach in automatic sequence-based test generation, it is explained in more detail. First, it defines several branch types. In a CFG, if a node n is only reachable by a subset of outgoing edges of a node n_i, then n is control dependent on n_i. Note that in [115] the term "in the scope of control influence" is used rather than control dependency. A branch $b = (n_i, n_j)$ is a critical branch with respect to n, if n is control dependent on n_i and no path from n_j to n exists. Branch b is a required branch with respect to n, if n is control dependent on n_i and an acyclic path from n_i to n exists including b. Branch b is a semi-critical branch with respect to n, if n is control dependent to n_i and no acyclic path from n_i to n exists which includes b. Branch b is a non-essential branch with respect to n, if n is not control dependent on n_i. Figure 2.20 exemplifies those branch types on the example of the iterative factorial method.

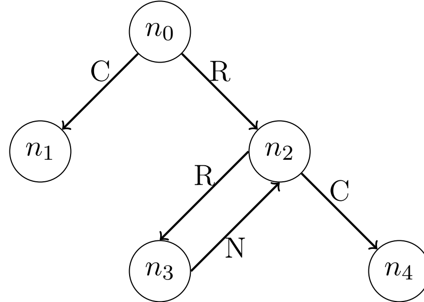


Figure 2.20: Branch types with respect to node n_3 as defined in [115]. The types are Critical, Semi-Critical, Required, Non-essential.

As an example, in order to maximize the structural coverage, node n_3, i.e. the body of the while-loop is targeted. The program is executed with a random input and the path of that execution is tracked. If the current branch is a non-essential or a required branch, the program execution is continued through that branch. If the current branch is a critical or a semi-critical branch, the program execution is terminated and hill climbing is used to find a new input. In contrast to Baresi et al. (2010) [39], Korel (1990) [115] uses a more sophisticated formula to measure the distance to branches. It is altered by Tracey et al. (1998) [191] and depicted in Table 2.4.

The distance $d_{\min(b,T)}$ is the minimum distance of branch b for all tests in test set T following table 2.4. It is altered by EVOSUITE to the following formula:

$$d(b,T) = \begin{cases} 0 & \text{if the branch has been covered} \\ v(d_{\min(b,T)}) & \text{if the predicate has been executed at least twice} \\ 1 & \text{otherwise} \end{cases} \tag{2.6}$$

Here $v(x)$ is a function which converts the distance to $[0,1]$ with $v(x) = x/(x+1)$. A predicate has to be executed at least twice in order to avoid circular behavior of the branch distance.

Table 2.4: Branch distance as defined by Tracey et al. (1998) [191]. K is a constant with $K > 0$.

condition	branch distance
boolean	`if TRUE then` 0 `else` K
$a = b$	`if` $\lvert a-b \rvert = 0$ `then` 0 `else` $\lvert a-b \rvert + K$
$a \neq b$	`if` $\lvert a-b \rvert \neq 0$ `then` 0 `else` K
$a < b$	`if` $a-b < 0$ `then` 0 `else` $(a-b)+K$
$a \leq b$	`if` $a-b \leq 0$ `then` 0 `else` $(a-b)+K$
$a > b$	`if` $b-a < 0$ `then` 0 `else` $(b-a)+K$
$a \geq b$	`if` $b-a \leq 0$ `then` 0 `else` $(b-a)+K$
$a \vee b$	$min(cost(a), cost(b))$
$a \wedge b$	$cost(a) + cost(b)$
$\neg a$	Negation is moved inwards and propagated over a

The final fitness function is defined as follows:

$$fitness(T) = |M| - |M_T| + \sum_{b_k \in B} d(b,T) \tag{2.7}$$

Here M are the methods of the SUT and M_T are methods of the SUT executed by test set T. Note that this fitness function targets all branches at once which is called "whole test suite generation" (WS). Fraser and Arcuri (2013) [82] claim that this strategy outperforms a single branch strategy using the described branch distance adapted by the *approach level* [203]. A more detailed study [165] backs those results and stresses that those cases, where the single branch target outperforms the whole test suite generation, are special cases rather than general deficiencies of the whole test suite approach. Note that here test suites are denoted as test sets. In addition, the EVOSUITE tool does not only contain the described fitness function and an overall well balanced setup for a genetic algorithm, but several other features which promote the generation of test cases. For instance, the generation of strings and primitives depend on the strings and primitives that are used in the SUT as constants. The usage of generic types is supported by a custom type system. Overall, from the various options EVOSUITE offers, some also may lead to a worse result than the default settings which are strongly recommended [82].

Rojas et al. (2016) [165] enhance the whole test suite generation (WS) approach by an archive. If a new test is discovered, which covers an at that time uncovered branch, the branch and the test are added to an archive. All branches which are contained in that archive are no longer targeted by the fitness function. The fitness function in the whole suite archive approach is:

$$fitness(T, B) = \sum_{b \in B \setminus C} d(b,T), \tag{2.8}$$

where C is the archive, B are all branches in the SUT and T the test set. Rojas et al. (2016) [165] claim that the whole suite archive (WSA) approach outperforms the original WS approach.

The tool GAMDR [28] uses the same fitness function as EVOSUITE. Where EVOSUITE uses the traditional mutation operators *remove*, *change* and *insert* to mutate single randomly

chosen test cases, GAMDR randomly chooses reached but not covered branches. The predicates of the chosen branches are analyzed and those statements in the test case are identified which have an impact on the predicates. These statements are then removed or changed or new statements are inserted with a probability of 1/3 for all three mutation operators. Aburas and Groce (2016) [28] claim that GAMDR achieves higher branch coverage than EVOSUITE for complex programs which are hard to cover by tests generated by an automated test generator.

2.11.4.2 Multi and Many Objective Genetic Algorithms

[148, 149] change the WS [82] and WSA [165] approaches to a multi objective formulation. That is, the fitness function does no longer combine all targeted branches by summing up all branch distances, but all branches are targeted individually. The fitness function is enhanced to a fitness vector. Without loss of generality, it is assumed for the remainder of this thesis, that all objectives in the fitness vector are to be minimized by the multi objective algorithm. As there is no longer one distinct value per solution to compare, the concept of dominance is introduced in multi-objective genetic algorithms [209]:

Definition 2.4 *Given two solutions x, y and their fitness vectors $f(x), f(y) \in R^n$, x dominates y, denoted as $x \prec y$, if and only if $\forall i \in \{1, 2, ...n\} : f_i(x) \leq f_i(y)$ and $\exists j \in \{1, 2, ...n\} : f_j(x) < f_j(y)$ [209].*

This dominance is also called Pareto dominance. A solution x^* is Pareto optimal if there does not exist another solution x that dominates it. The union of all Pareto optimal solutions is called Pareto set (PS): $PS = \{x \mid \nexists\, y, y \prec x\}$ and the corresponding objective vector set of the PS is called the Pareto front [119]. Figure 2.21 shows a graphical representation of Pareto dominance. Given is a SUT with only two branches and five fitness vectors from the five corresponding solutions, i.e. test sets, $\{A, ..., E\}$. All test sets in the grey rectangle (A and B) are dominated by C because C is better for both objectives f_1 and f_2. Test set C does not dominate D. C is closer to cover f_1 but worse than D on the other test target f_2. Similarly, C does not dominate E. Thus, C,D and E are non-dominated by any other test set while A and B are dominated by either C or D.

Given the dominance relation, multi objective optimization problems may lead to a set of Pareto optimal solutions, i.e., solutions which are all optimal in their position of the objective space. Especially when the number of objectives is greater than three, this may lead to the dominance resistance phenomenon, which denotes the incomparability of solutions caused by an increasing proportion of non-dominated solutions [119].

The recent literature differentiates between multi objective (up to four optimization goals) and many objective (above three optimization goals) algorithms. Nonetheless, many objective algorithms are often presented as a subset of multi objective algorithms. The genetic algorithms used for the performance test generation in this thesis use a many objective fitness function, which is described in Section 5.2.2.5. Detailed surveys on many objective genetic algorithms can be found in [119, 199]. As a basis algorithm for the most recent approaches using multi and many objective optimization in automated structural test generation, the non dominated sorting genetic algorithm (NSGA-II) [75] is firmly explained here. A pseudo-code listing is given in Algorithm 2.

As the simple single objective genetic algorithm, the NSGA-II starts with an initial random set of solutions. Until the stopping condition, like a certain time limit, is reached, the next generation of solutions is evolved as follows. At the begin a new offspring is generated and

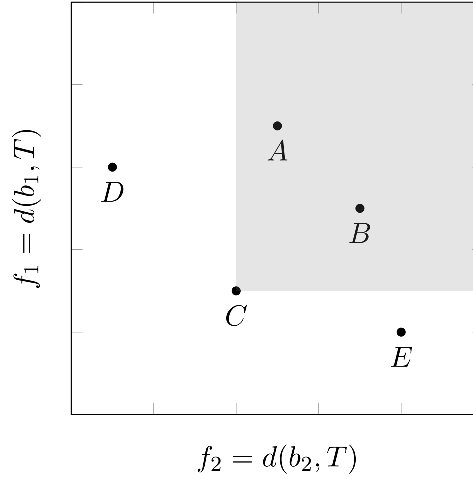


Figure 2.21: Visualization of Pareto dominance. C dominates all solutions in the grey area.

Input : Stopping condition C, Fitness function δ, Population size p_s, Selection function s_f, Recombination function r_f, Recombination probability r_p, Mutation function m_f, Mutation probability m_p

Output: Population of optimized individuals P

```
1  t ← 0;
2  P_t ← GenerateRandomPopulation(p_s);
3  while ¬C do
4  |  Q_t ← GenerateOffspring(r_p,r_f,m_p,m_f,P_t) R_t ← P_t ⋃ Q_t ;
5  |  𝔽 ← FastNonDominatedSort(R_t);
6  |  P_{t+1} ← {} ;
7  |  d ← 1;
8  |  while |P_{t+1}| + |𝔽_d| ≤ p_s do
9  |  |  CrowdingDistanceAssignment(𝔽_d);
10 |  |  P_{t+1} ← P_{t+1} ⋃ 𝔽_d;
11 |  |  d ← d + 1;
12 |  end
13 |  Sort(𝔽_d) ; // according to the crowding distance
14 |  P_{t+1} ← P_{t+1} ⋃ 𝔽_d with size p_s − |P_{t+1}|;
15 |  t ← t + 1;
16 end
```

Algorithm 2: Pseudo-code for the NSGA-II.

then added to the current generation (line 4-5) using mutation (e.g. adding or removing test cases) and recombination (e.g. rearranging test sequences). Afterwards, the parents of the next generation are selected preferring non-dominated solutions in the current population (line 6). The next generation is then created selecting the best fitting solutions from the current generation. The *crowding distance* is used in order to make a decision about which test cases to select: non-dominated test cases that are far away from the rest of the population have

higher probability of being selected (line 9-15). The *fast dominated sort* is used to preserve solutions from the current approximation of the pareto optimal set in the next generation (lines 6 and 15).

Although the NSGA-II is more a multi objective than a many objective optimization algorithm, it is often used as a base for many objective approaches and compared to them [119, 199]. For instance, to maintain a wider spread set of test sets, the ϵMOEA [76] uses the ϵ dominance relation introduced by Laumanns et al. (2002) [117]. In contrast, the MOEA/D [120] does not use a classic selection operator. Instead, it decomposes a multi objective into several single objective sub optimizations. It uses aggregation coefficient vectors to optimize each objective for its own. This is done in order to achieve a wider spread in the solutions and to optimize the computation time.

The *many objective sorting algorithm* (MOSA) [148] is explicitly designed for the generation of test sets with a high structural coverage. Using the branch distance and the approach level, all branches are targeted individually. The approach level is designed to avoid local optima by incorporating all predicates that are evaluated by the given input when reaching the targeted branch. It benefits those predicates which are evaluated targeted, i.e. predicates that are needed to reach the desired branch [204]. Using those objectives, MOSA additionally changes the original NSGA-II in four ways:

1. The non-dominated sort selection (line 6 in Algorithm 2) is altered by a custom *preference sorting* which prefers test cases with uncovered branches.

2. When the algorithm has to decide whether two test cases dominate each other or not (line 6), the dominance comparator in MOSA only iterates over uncovered targets and stops immediately if it finds two uncovered targets for which the two test cases do not dominate each other.

3. Another crowding distance assignment is used (line 10). The test cases which have a higher distance to the rest of the population are given a higher probability of being selected. The function is adapted from [113].

4. A second population, called *archive*, is used to preserve test cases which satisfy previously uncovered targets as candidates for the final test set.

A recent extension of MOSA is DynaMOSA [149] which dynamically selects the optimization targets. That is, with respect to the general goal of test cases with a high structural coverage, DynaMOSA uses the control dependency of the targets. At the start, DynaMOSA only selects those targets that are free of control dependencies. This set of targets is then updated with respect to the targets that were included during the generation of the new offspring (line 5 in Algorithm 2). That is, all uncovered targets that are control dependent on the newly covered targets are added to the target set. Panichella et al. (2017) [149] claim, that DynaMOSA has a significantly better performance in terms of achieved coverage of the generated test cases than MOSA and WSA. In a recent study, Campos et al. (2017) [59] support these conclusions.

2.11.4.3 Benchmarking Search Based Approaches

The annual *Java Unit Testing Tool Competition* [42, 166, 167, 17] compares the effectiveness of JUnit test generators in terms of achieved coverage (instruction, branch, ratio of killed

mutants), the faults that were found, the test suite size and a given time limit. The score also incorporates a penalty for uncompilable and unstable, called flaky, test cases. A summary of the score values is given in Table 2.5d. Note that here random testing is considered to be a part of search based testing and that the score values of different years are not directly comparable as the calculation of the score changes over time. Nonetheless, it provides a good overview of the current achievements in search based software testing and automatic test generation. For those years for which a manual test is given it outperforms the automatic generators. Only in 2015 is EVOSUITE outperformed by another test generator, GRT.

Table 2.5: Results of the SBST Unit testing tools competitions [17]. Results of different years are not comparable to each other. Different results found for EVOSUITE at the FITTEST 2013.

(a) Results of the SBST 2013 Java Unit Testing Tool Competition [25, 42]

Tool	RANDOOP	EVOSUITE	T2
Score	101.8129	156.9559	50.4938

(b) Results of the FITTEST 2013 Java Unit Testing Tool Competition [41]

Tool	RANDOOP	EVOSUITE		T3	Manual
Score	93.45	205.26 [83]	199.57 [158]	144.98	210.45

(c) Results of the SBST 2015 Java Unit Testing Tool Competition [166]

Tool	RANDOOP	EVOSUITE	T3	JTExpert	MOSA	GRT	Manual
Score	93.45	190.64	186.15	159.16	189.22	203.73	210.45

(d) Results of the SBST 2016 Unit Testing Tool Competition [167]

Tool	RANDOOP	EVOSUITE	T3	JTExpert
Score	747	1127	978	931

2.11.4.4 Genetic Algorithms in GUI Test Generation

Search based testing is also successfully used to generate GUI tests [60]. Static approaches, such like GUITAR [142] build the GUI model using a ripping procedure before generating the test cases. Still, GUITAR is more of an adaptable framework rather than a unique approach on GUI test generation. Dynamic approaches build the GUI model and generate the test cases dynamically. EXYST [93, 94] uses the experiences made with EVOSUITE to generate small GUI test sequences with a high code coverage. Pidgin Crasher [71] is developed to find GUI sequences which cause system crashes. Menninghaus et al. (2017) [135] compare several multi-objective genetic algorithms (NSGA-II [75], PESA2 [68] and SPEA2 [213]). They generate GUI tests with the same general settings but are not able to distinguish the three genetic algorithms in terms of achieved test coverage and conclude that more effort on configuration and optimization is needed in order to benefit from one genetic algorithm or the other.

2.11.4.5 Analyzing Evolutionary Algorithms

In this thesis, single and multi-objective genetic algorithms are used to create test sets with a high structural coverage. The different approaches are not only compared in terms of the results, i.e. the coverage of the generated test cases, but in the way they approach their goal. To quantify the several genetic algorithms and their settings, the evolvability metrics described in this section are used.

Evolvability describes the capability of a genetic algorithm to create a child generation that has a better fitness than the parent generation. As a random approach is also capable to find better solutions, the distributions of parent and children populations need to be taken into account. While the final result of a genetic algorithm (e.g. the coverage of the generated test sets) describe the general capability of the GA to generate the desired solutions, the evolvability allows a qualitative analysis on a local level [32].

For the remainder of this section, let $\{f_i^*\}_{i=1}^n$ be the sequence of the best fitness values after each iteration $1 \leq i \leq n$. The value of the best fitness value is monotonically increasing with $f_i^* \neq f_{i+1}^* \Leftrightarrow f_i^* < f_{i+1}^*$. Usually, genetic algorithms preserve the best solution which is called elitism. A sequence of fitness values may be transformed into a binary sequence $\{b_i\}_{i=1}^{n-1}$ with

$$b_i = \begin{cases} 0 & \text{if } f_i^* = f_{i+1}^* \\ 1 & \text{otherwise.} \end{cases} \tag{2.9}$$

Change Rate (CR) The change rate (CR) represents the relation between the number of iterations with an increasing fitness value and the total amount of iterations [29]:

$$CR = \frac{\sum_{i=1}^{n-1} b_i}{n} \tag{2.10}$$

The CR does only quantify the relative number of iterations with increasing fitness. It does not analyze the sub-sequences of increasing or stagnating fitness values.

Population Information Content (PIC) In contrast to the CR, the population information content (PIC) [29] measures whether a genetic algorithm is more likely to follow only a single gradient or to explore different basins of attraction. It analyzes the binary sequence $\{b_i\}_{i=1}^{n-1}$ as a sequence of overlapping blocks of two bits. They determine whether the fitness is increasing after stagnation ($\{01\}$), stagnating after increasing ($\{10\}$), continuously stagnating ($\{00\}$) or increasing ($\{11\}$). The probability of encountering one of those sub-sequences is given by the number of times the sub-string is found divided by the total number of sub-sequences:

$$P_{\{b_i, b_{i-1}\}} = \frac{|\{b_i b_{i-1}\}|}{n-1} \tag{2.11}$$

The average amount of information contained in the binary sequence $\{b_i\}_{i=1}^{n-1}$ is then calculated as sum of the probabilities for a changing fitness. Note that only two sub-strings with a changing fitness exist: $\{01\}$ and $\{10\}$. Aleti et al. (2017) [29] sum the probabilities for each bit in the given bit-string which means that the number of iterations in a genetic algorithm influences the PIC. In contrast, with respect to the entropy by Shannon and Weaver (1949) [179], the PIC is defined as:

$$H = -P_{\{01\}} log_2 P_{\{01\}} - P_{\{10\}} log_2 P_{\{10\}} \tag{2.12}$$

The H value approaches zero if the sequence has no changes. The lower the value of H, the more irregular the fitness development. The general assumption of the PIC is, that a fitness development which undergoes improvements with interspersed non-improving iterations is more likely to explore different basins of attractions while continuous improvement indicates, that a single gradient is followed [29].

An overview of recent methods for fitness landscape analysis can be found in [157].

2.11.4.6 Additional Issues in Search Based Testing

In addition to the mentioned tools, Memon (2001) [131] focuses on coverage criteria for inter- and intra-component coverage. Yuan et al. (2011) [211] extend that idea with covering arrays [62] to unfold faults more goal-oriented by controlling the sequence length, the possible positions of events and certain combination of events. [36] investigate the best sequence length of tests regarding specific configurations. Fraser and Arcuri (2011) [81] stress the correct bloat control, with bloating being the disproportional quickly growth of the length of test sequences.

Sharma et al. (2010) [180] compare the general approach of sequence-based test generation to constraint-based test generation. Unlike constraint solvers that are mentioned with symbolic execution constraint-based test generation is used to generate complete data structures for testing. A constraint-based approach would generate a data structure on the basis of a given general definition of this structure. For instance, given the definition of a heap, a constraint-based approach would then produce as much heap structures as necessary to test the implementation fully. The tool Korat [127, 137] generates complex data structures based on Java predicates which then can be used for testing the implementation of these structures. Since in this thesis interchangeable test sequences are used, none of the approaches on constraint-based test generation is applicable.

All presented tools and approaches are guided by various sets of parameters. For each new class of SUTs encountered, the software tester has to optimize those parameters. Most tools are configured such that they give good results for the most common SUTs. Arcuri (2011) [36] focuses his research on the correct strategies to set the parameters to the desired optimum. He states that the default parameters of EVOSUITE always give good results but that manual parameter manipulation may also lead to worse results.

2.11.5 Summary

An overview of all tools reviewed in this section is given in Tables 2.6 and 2.7. It should be noted that none of the reviewed tools is designed to generate interface based test cases as required by this thesis. Nonetheless, those tools which are capable of generating test cases based on interfaces after some minor alterations are evaluated in more detail in Section 5.2.1. Most notably, while some older tools like KLEE are still maintained [14], the most recent tools use a search based approach including random search. Genetic algorithms which use the branch distance approach and either a whole suite or many objective approach which targets all branches are favored. Due to the danger of path explosion, concolic testing seems not to be of recent interest. Thus, recent automatic test generation approaches seem to prefer a guided search based on randomness instead of proven complete test generators. Those approaches which are based on guided randomness always face the pitfall that special cases may be created which are overseen by the guiding algorithms. Several new automatic test generators which only use a user given interface for the sequence generation are created throughout this thesis.

Table 2.6: Summary of automatic test generation, sorted by publication date for the years 2004-2010. Unnamed tools are denoted with -.

Name	Approach	Language	Input	Output	Source	Year	Notes
JCrasher	Random	Java	source code	JUnit tests	[70]	2004	
Jartege	Random	Java	JML model and byte code	test methods	[145]	2005	JML specification used to avoid invalid test cases
PathCrawler	Concolic	C	source code and preconditions	test inputs	[206]	2005	single test inputs, all paths criterion limited to k iterations per loop
DART	Concolic	C	source code	test inputs	[88]	2005	single test inputs
CUTE / jCUTE	Concolic	C, Java	source code	JUnit tests	[178, 177]	2005	
-	Model	XML	UML Activity Diagram	GUI tests as XML-files	[197]	2006	creates functional tests
RANDOOP	Random	C, Java	byte code and properties to check	JUNIT tests	[146]	2007	
EXE	Concolic	C	source code	failing test inputs	[58]	2008	single test inputs
KLEE	Concolic	C	source code, annotations	test inputs	[57, 14]	2008	redesign of EXE
Barad	Concolic	Java	source code / GUI	Swing GUI tests	[86]	2008	
-	Model	Java	source code, UML Activity Diagram	coverage statistics	[63]	2009	tests are created and executed within the tool
WISE	Concolic	Java	byte code	test inputs	[56]	2009	creates tests with worst case complexity for single method invocations
-	Model	Java	source code, GUI specification	GUI tests as XML-files	[210]	2010	based on GUITAR [142]
TestFul	SBST	Java	source code	JUnit tests	[39]	2010	combines genetic algorithms and hill climbing

Table 2.7: Summary of automatic test generation, sorted by publication date for the years 2011-2017. Unnamed tools are denoted with -.

Name	Approach	Language	Input	Output	Source	Year	Notes
EVOSUITE	SBST	Java	byte code	JUnit tests	[81]	2011	whole suite (WS) approach
ACTIVe	Concolic	Java	source code	Android GUI tests	[34]	2012	
EXYST	SBST	Java	byte code	JUnit tests	[93]	2012	based on EVOSUITE, but from GUI perspective
Collider	Concolic	Java	source code, UI model, targets	Android GUI tests	[109]	2013	
T3	Random	Java	byte code	JUnit tests	[158]	2013	supports and uses closures and Java 8 streams
MOSA	SBST	Java	byte code	JUnit tests	[148]	2015	based on EVOSUITE, adaption of NSGA-II used
JTExpert	Random	Java	source code	JUnit tests	[170]	2015	guided random generation
GRT	Random	Java	byte code	JUnit tests	[126]	2015	guided random generation
-	SBST	Java	byte code	JUnit tests	[165]	2016	based on EVOSUITE, whole suite archive (WSA) approach
GAMDR	SBST	Java	byte code	JUnit tests	[28]	2016	WS approach, mutates reached but not covered branches
GUItoolkit	SBST	Java	source code	Swing GUI tests	[135]	2017	uses many objective algorithms, hill climbing and simulated annealing
DynMOSA	SBST	Java	byte code	JUnit tests	[149]	2017	based on MOSA, adds archive

Including those which are specialized for the generation of test cases for high-dimensional spatio-temporal index structures.

2.12 Conclusions for this Thesis

In order to create a high-dimensional spatio-temporal index structure for discretely changing planing data and a mostly automated approach to analyze the performance behavior of that method in contrast to existing ones without the usage of an artificial benchmark, the following conclusions from the previous sections should be kept in mind:

- The most prominent spatial access method, the R^*-tree, suffers from the *curse of dimensionality* as well as the only structure which is able to handle discretely changing spatial data well, the R^{ST}-tree.
- High dimensional access methods do not support spatio-temporal data. An extension of the R^*-tree, the X-tree may be altered such like the R^{ST}-tree but it is heavily system dependent and may not be suitable for in-memory use.
- No benchmark exists for the desired spatio-temporal or high-dimensional requirements, such that a new benchmark has to be created.
- The definition of benchmarks and workload generation and the requirements of *good* benchmarks suggest a strong community or another possibility of verification, for instance by real-world data which is not applicable for a complete new type of structure.
- Performance measurement and comparison is always system specific and when using Java, one has to keep in mind that both, the JIT compilation and the measurement using bytecode instrumentation may affect the general conclusion of an evaluation.
- With the general goal for automated performance test generation and comparison the necessity of user input should be minimized. The system created in this thesis uses structural coverage.
- As no flexible coverage recording tool for Java programs exists, a new one is required.
- For the desired automated test generation, search based test generation is of the most recent interest. Especially the use of either a single objective but whole suite approach using a genetic algorithm or a multi objective genetic algorithm.

Chapter 3

The Spatio-Temporal Pyramid Adapter

Section 2.2.3 shows that no spatio-temporal indexing technique exists which handles high-dimensional data well and no high-dimensional indexing technique exists which is able to store *now*-relative data. In this chapter, a new spatio-temporal high-dimensional indexing technique, the Spatio-Temporal Pyramid Adapter (STPA) which performs well on both data types, is presented. It is partly published in [134]. It is shown that the most promising existing approach for the indexing of spatio-temporal high-dimensional data, the X-tree, is not a sufficient base for the index structure. After presenting the design of the STPA (Section 3.2) and its implementation (Section 3.4) in detail, its query performance is evaluated (Section 3.5) using the workload generator for spatio-temporal data according to Saltenis and Jensen (1999) [172]. An evaluation of the STPA based on automatically generated test sets is provided in Section 6.2.1.

3.1 Analysis of the X-Tree

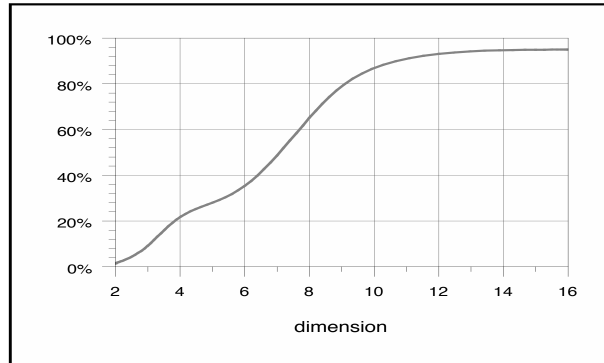
From the review of existing approaches in high-dimensional indexing, the X-tree [46] seems to be the most promising for the adaption of spatio-temporal data. In addition, it is one of the most often cited index structures[1]. The probably best technique for the indexing of discretely changing spatio-temporal data, the R^{ST}-tree, is based on the R^* [44]. As the X-tree is also based on the R^*-tree, the alterations made by the R^{ST}-tree may also be applied on the X-tree. This section evaluates, why the X-tree is not chosen as a basis for the part of handling high-dimensional data in the new technique.

Instead of the overlap defined by Beckmann et al. (1990) [44] (Equation 2.1), Berchtold et al. (1996) [46] use the following formular to compute the overlap in an R-tree:

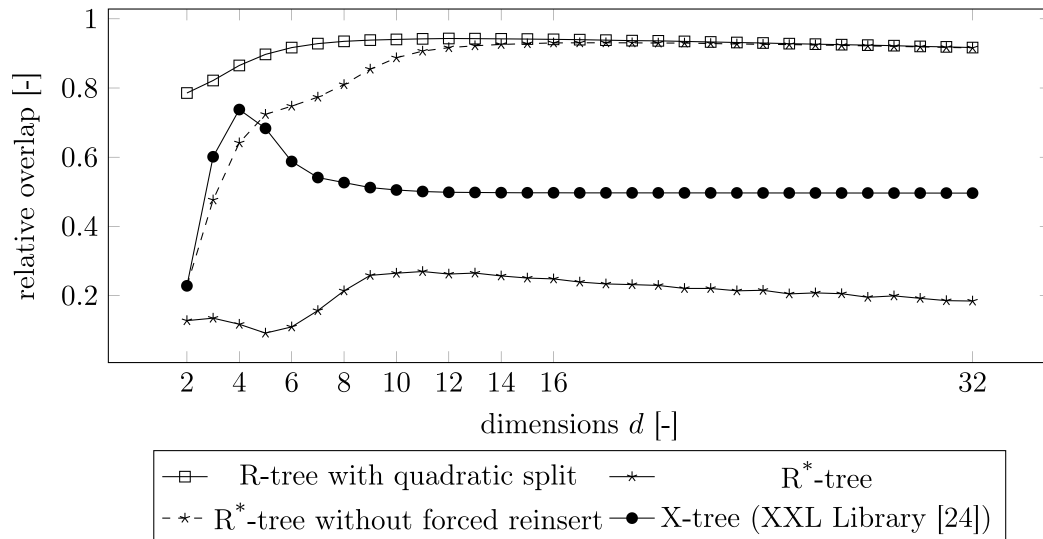
$$overlap = \frac{area\left(\bigcup_{i,j \in \{1,..,n\}, i \neq j} R_i \cap R_j\right)}{area\left(\bigcup_{i \in \{1,..,n\}} R_i\right)} \tag{3.1}$$

[1] 1989 citations according to Google Scholar on May, 4th 2017

with R_i being the minimum bounding rectangle (MBR) of a directory node and function *area* computing the area for two-dimensional and the volume for higher dimensional MBRs. Dividing this overlap by the number of directory nodes in the corresponding R-tree results in the relative overlap $o \in [0, 1]$. According to Berchtold et al. (1996) [46], a high value of o indicates a probably poor query performance. Every time a query is performed, all directory nodes have to be traversed, whose MBRs are affected by the query. A high overlap between the directory nodes increases the number of nodes that have to be traversed in contrast to a low overlap between the directory nodes.



(a) Original computed relative overlap of directory nodes (y-axis) of the R*-tree as published in [46].



(b) Mean value of the relative overlap of directory nodes according to Berchtold et al. (1996) [46] for several R-tree variants and implementations.

Figure 3.1: Comparison of the relative overlap of directory nodes according to Berchtold et al. (1996) [46]. The upper plot shows the original computation, the lower plot shows the computation done in this thesis.

Figure 3.1 shows the original evaluation of the overlap on an R-tree which contains uniformly distributed data (3.1a) and the evaluation of the overlap of R-, R*- and X-tree along

with the R*-tree without the forced reinsert being performed instead of a split (3.1b). Since the setup for the original evaluation is lost[2], the parameters for the new evaluation are set in order to reproduce the general behavior of the original evaluation. For each dimension $d \in \{2, .., 32\}$ 1000 test cases are computed, each with an index that contains 10000 uniformly distributed points in $[0, 1]^d$. The points are inserted one by one into an empty index. The plots show the mean relative overlap for each dimension. The relative overlap of the R*-tree seems to reflect the original evaluation. The dent between the dimensions 4 and 6 seems to be misplaced, but due to the uncertainty of the R*-tree and the fact that the original evaluation may not display the mean but a single test set, it is very likely that the index used for the original evaluation is the R*-tree without the forced reinsert as an overflow technique.

Since the original implementation is lost[3], the implementation provided by the eXtensible and fleXible Library (XXL) [24] for Java is used. To the knowledge of the author, it is the only available implementation of the X-tree. In addition, it is recommended by the authors of the original paper[4]. Figure 3.1b shows that the X-tree performs better in terms of the relative overlap but is clearly outperformed by the R*-tree. More importantly, the relative overlap does not rise for more than 10 dimensions, but is slowly declining with an increasing number of dimensions. The question arises whether the relative overlap is a suitable metric to describe the decreasing query performance when the number of dimensions increases (see Section 2.2.3). The same conclusions can be made for the weighted overlap which is also proposed in [46]. In addition, the X-tree is based on system specifics such as the CPU time needed to read a block from the hard disk. Such parameters have no counterpart for an index which works in memory.

3.2 Design of the Spatio-Temporal Pyramid Adapter

The new STPA technique needs to incorporate both, spatio-temporal and high-dimensional data. How this goal is approached is shown in this section.

Four different possible approaches for the generation of a new high-dimensional spatio-temporal index structure exist: (1) combining an existing spatio-temporal and an existing high-dimensional indexing technique, (2) extending an existing spatio-temporal technique to support high-dimensional data, (3) extending an existing high-dimensional technique to support spatio-temporal data, (4) creating a completely new structure.

Following from the indexing approaches discussed in Section 2.2 and the conclusions for the X-tree in Section 3.1, (1) seems, to the knowledge of the author, not applicable. For (2), a spatio-temporal indexing technique has to be altered such that it does not face the *curse of dimensionality*. Since the overlap seems not to be a cause of a decreasing query performance with an increasing number of dimensions (Section 3.1), two other causes can be identified. First, if the high-dimensional data points are stored inside the directory nodes and these nodes are stored on disk blocks, which is the most common technique, the number of entries which can be stored per node naturally decreases with the number of dimensions. Secondly, usually, if a data point is computed, each of its dimensions has to be computed. With an increasing number of dimensions, the computation time for each data point also

[2] Personal contact with the authors. Stefan Berchtold: Nov. 20th 2014, Daniel A. Keim: May 11th 2015, H.-P. Kriegel: May 12th 2015

[3] ibid.

[4] ibid.

increases. To the knowledge of the author, all spatio-temporal indexing techniques face the *curse of dimensionality* and the described problems cannot be eliminated without making the structures incapable of handling spatial data, i.e. hyper-rectangles. Much more promising is (3), especially in contrast to a completely new structure (4). As a matter of fact, a high-dimensional structure should not face the *curse of dimensionality*. Using it as a basis for the new structure, one has to find an appropriate representation for the *now*-value. Section 2.1.2.1 discusses the different possible representations of *now* in an index structure. Note that the concrete temporal and especially spatio-temporal model is not of concern as the index should support all kinds of spatio-temporal data models. The choice of one representation depends on the underlying high-dimensional technique. To the knowledge of the author, none high-dimensional technique except for the X-tree does support spatial data. Therefore, the chosen indexing technique has to be extended to be capable of handling interval data, i.e. storing a begin and an end value in each dimension per object instead of a single value. By now, two steps are required: Finding an appropriate mapping of spatial data onto point data and incorporating *now* in that mapping.

As a matter of fact, a high-dimensional structure should not have any problems indexing objects with d or even $2d$ dimensions. The new STPA maps spatial data onto point data by doubling the dimensionality of the points. Given a d-dimensional rectangle with each dimension containing two values, begin ($\vdash$) and end ($\dashv$), the $d \vdash$ values of each dimension are mapped to the first d values of the point and the $d \dashv$ values are mapped on the second d values of the point. Mapping the d-dimensional rectangles onto $2d$-dimensional points permits the usage of the possible representations of *now* as described in Section 2.1.2.1. In particular, the most efficient approach, the POINT approach, relies on data structured in intervals. Using a value outside the data space or the maximum or minimum value to represent *now* would shift the converted points to the borders of the data space and make queries much more complex or always lets them cover clearly more space than needed by the actual query. Some high-dimensional techniques use one or more constant data points to represent a set of high dimensional points and separate them by the data point the point has been clustered to and the distance to that point (Section 2.2.3). Using one of those techniques and excluding the *now*-value from the clustering and distance computation would not expand the *now*-relative queries. In contrast, the *now*-relative points are then more likely attracted by the constant data points which already are covered by most of the query rectangles. For instance, a two-dimensional point $p = (1, now)$ would be clustered only with respect to the first value (1) which makes it more likely that p is near to a constant data point. From the given techniques, the Pyramid Technique [46] is chosen because it is based on the R$^+$-tree which is already supported by most of the current databases and its extensions make it possible to adapt dynamically to different distributions of data (see the Extended Pyramid Technique [46] or the P$^+$-tree [212]).

As a reminder, the Pyramid Technique divides the d-dimensional data space into $2d$ pyramids. A d-dimensional point is mapped onto a decimal value. The digits to the left of the decimal denotes the number of the pyramid at which the point has the lowest height, i.e. the distance along that pyramid's center axis to the center of the data space. The digits to the right of the decimal point are the actual height. It is assumed that the complete d-dimensional data space lies in $[0, 1]^d$.

Figure 3.2 displays the conversion of *now*-relative rectangles if the Pyramid Technique is used as underlying technique and *now* is mapped to the center of the data space. It assumes a two-dimensional data space with one *now*-relative dimension (d_0). For the remainder of this thesis, the data space $[0, 1]^d$ is always assumed. For the static rectangle R_0, the conversion

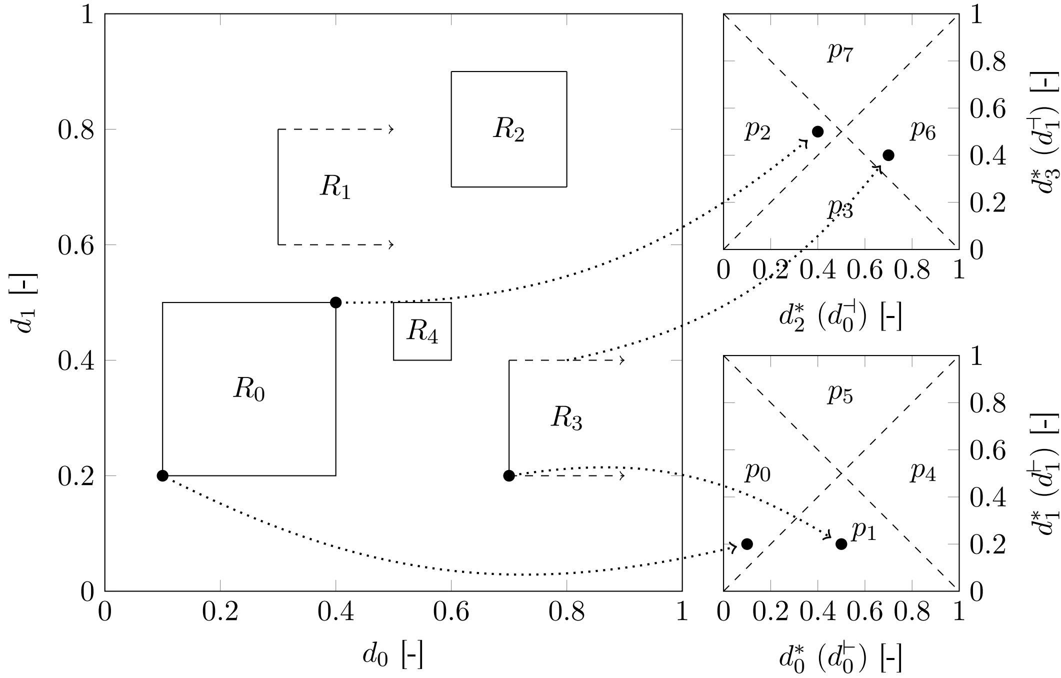


Figure 3.2: Example conversion of spatio-temporal rectangles (left) to pyramid values (right). The two graphics on the right are both a two-dimensional view on the very same four dimensional space from orthogonal perspectives. In order to keep the numeration of the pyramids comprehensible, the dimensions of the $2d$-space are numbered from 0 to $2d-1$.

works as follows: The beginnings of the dimensions d_0 (0.1) and d_1 (0.2) are mapped to the dimensions d_0^* and d_1^* and the ends at dimensions d_0 (0.4) and d_1 (0.5) are mapped to the dimensions d_2^* and d_3^* respectively. The 2-dimensional data space is divided into eight pyramids. The pyramid in which the mapped point has the highest distance is p_0 and the distance along axis d_0^* is 0.4 which results in a pyramid value of 0.4 for R_0. The conversion for the other rectangles works analogously, except for the *now*-relative part. For instance, the *now* values in R_3 are converted to the center of the data space in the respective dimension, which is always 0.5 for the original Pyramid Technique. Doing so, the interval on dimension d_0 is converted to $(0.7, 0.5)$, with 0.7 being the value of the begin and 0.5 the current value of *now*. Since an interval must never have a start value greater than its end value, both values have to be swapped with conversion. As imagined, the conversion of *now* leads to the effect that the *now* value does not affect the computation of the corresponding pyramid value as the distance for the dimension containing *now* is 0. Note that the Extended Pyramid Technique [47] does not use 0.5 as center in each dimension, but an approximated median of all inserted points in each dimension. The center would therefore dynamically shift during the usage of the STPA.

The very basis of the STPA for the insertion of a given spatio-temporal MBR R_i of the data $data_i$ with d dimensions is:

1. Map R_i to the $[0,1]^d$ space by dividing each value by the maximum in the corresponding dimension.

2. Convert the rectangles $R_i = ((r^{\vdash}_{i,0}, ..., r^{\vdash}_{i,d-1}), (r^{\dashv}_{i,0}, ..., r^{\dashv}_{i,d-1}))$ to the points $P_i = (r^{\vdash}_{i,0}, ..., r^{\vdash}_{i,d-1}, r^{\dashv}_{i,0}, ..., r^{\dashv}_{i,d-1})$, setting *now* to the current median value in the corresponding dimension.

3. Compute the Pyramid values p_i of P_i according to Berchtold et al. (1998) [47].

4. Store p_i as key with a pointer to the original data $data_i$ as value in a B$^+$-tree, as suggested by Berchtold et al. (1998) [47].

5. (Extended Pyramid Technique:) If the distance of the current used median and the actual median exceeds a given threshold, rebuild the index with the actual median [47].

The lookup, deletion and update of the STPA work analogously to the insertion procedure. The original object is simply converted by the same procedure as for the insertion and then the lookup, deletion or update is computed on the underlying B$^+$-tree.

3.3 Querying

The underlying B$^+$-tree only supports one-dimensional range queries. In addition, the conversion of *now* to a fixed value in conjunction with the nonetheless constantly ongoing time requires complex manipulations of any spatio-temporal query performed on data stored with the STPA. Each of these conversion steps is presented in this section. As an overview, one has to consider the following manipulations, given a spatio-temporal query consisting of a query type and a d-dimensional query rectangle q in the $[0, 1]^d$ space.

1. Identify the interval query types affected by the spatio-temporal query.

2. Convert the spatio-temporal query to the identified interval queries in every dimension.

3. Convert the interval queries to two-dimensional region queries on the mapped pyramid-space.

4. Adapt the region queries if they contain *now* or if they match the current value of *now* (now^*).

5. Merge the resulting d two-dimensional region queries to one $2d$-dimensional query in the pyramid space.

6. Convert the query to up to $4d$ range queries on the underlying B$^+$-tree according to the Pyramid Technique [47].

As the query is performed on the corresponding MBRs of each object, one has to note that every object in the result set is only a very good preselection and may contain false positives. Therefore, every spatio-temporal object in the result set has to be matched with the original query at the end of the query process to remove these false positives. Every conversion step is described in more detail in the following subsections. Figure 3.5 provides a complete example for a *now*-relative query on a two-dimensional space with one temporal (d_0) and one spatial (d_1) dimension.

3.3.1 (1) Identify Interval Query Types

Since the data is partitioned into intervals in each dimension, the STPA first needs to convert the query into interval queries. Allen (1983) [31] and Kriegel et al. (2001) [116] suggest 13 relationships between intervals which are visualized in Figure 3.3.

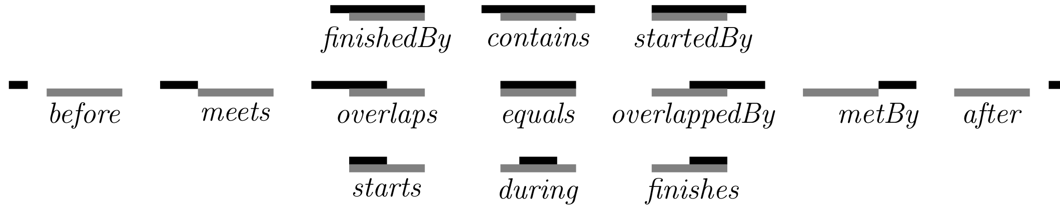


Figure 3.3: Thirteen general interval relationships according to Allen, Kriegel et al. (1983, 2001) [31, 116]. The gray line denotes the query interval, the black line a possible match to that query.

He et al. (2013) [97] combine these relationships to the eight fundamental relationships between two d-dimensional objects suggested by Egenhofer (1989) [78]. Likewise, the STPA determines which interval queries are affected by the given d-dimensional query and unites the ranges of these interval queries. In contrast to the approach given by He et al. (2013) [97], it is not efficient to split the query into several sub-queries and combine the results with logical operators like *AND* or *OR*. The STPA would either need to join the possibly large result sets of the sub-queries or track which elements already have been matched by a query. He et al. (2013) [97] suggest to use a flag for every element and every dimension to depict if an element has already been visited by a query in that dimension. This also means that this flag has to be reset after every query. Just like the join of the result sets of every sub-query in the first variant, this reset requires each queried element to be called again and causes a large overhead of I/O operations. Table 3.1 shows which interval query is affected by which of the fundamental relationships.

Table 3.1: List of the general interval relationships which are affected by the eight relationships between n-dimensional objects according to He et al. (2013) [97]. The *covers* and *coveredBy* interval relationships are considered to be special cases of the original *equals* relation between intervals with and without the borders [97].

Relationships between objects [78]	Interval relationships [31]
disjoint	*before*, *after*
meet	*meets*, *metBy*
overlap	*overlaps*, *overlappedBy*
equal	*equals*
contain	*contains*
contained	*during*
cover	*covers*
coveredBy	*coveredBy*
not affected	*finishes*, *finishedBy*, *starts*, *startedBy*

3.3.2 (2 + 3) Conversion to Region Queries

After identification of the query intervals to be used, the interval queries need to be mapped on the $2d$ pyramid space. Figure 3.4 shows how a one-dimensional interval query $(q^{\vdash}, q^{\dashv})$, where $q^{\vdash}$ is the begin and $q^{\dashv}$ is the end of the query interval, is converted into a two-dimensional region query.

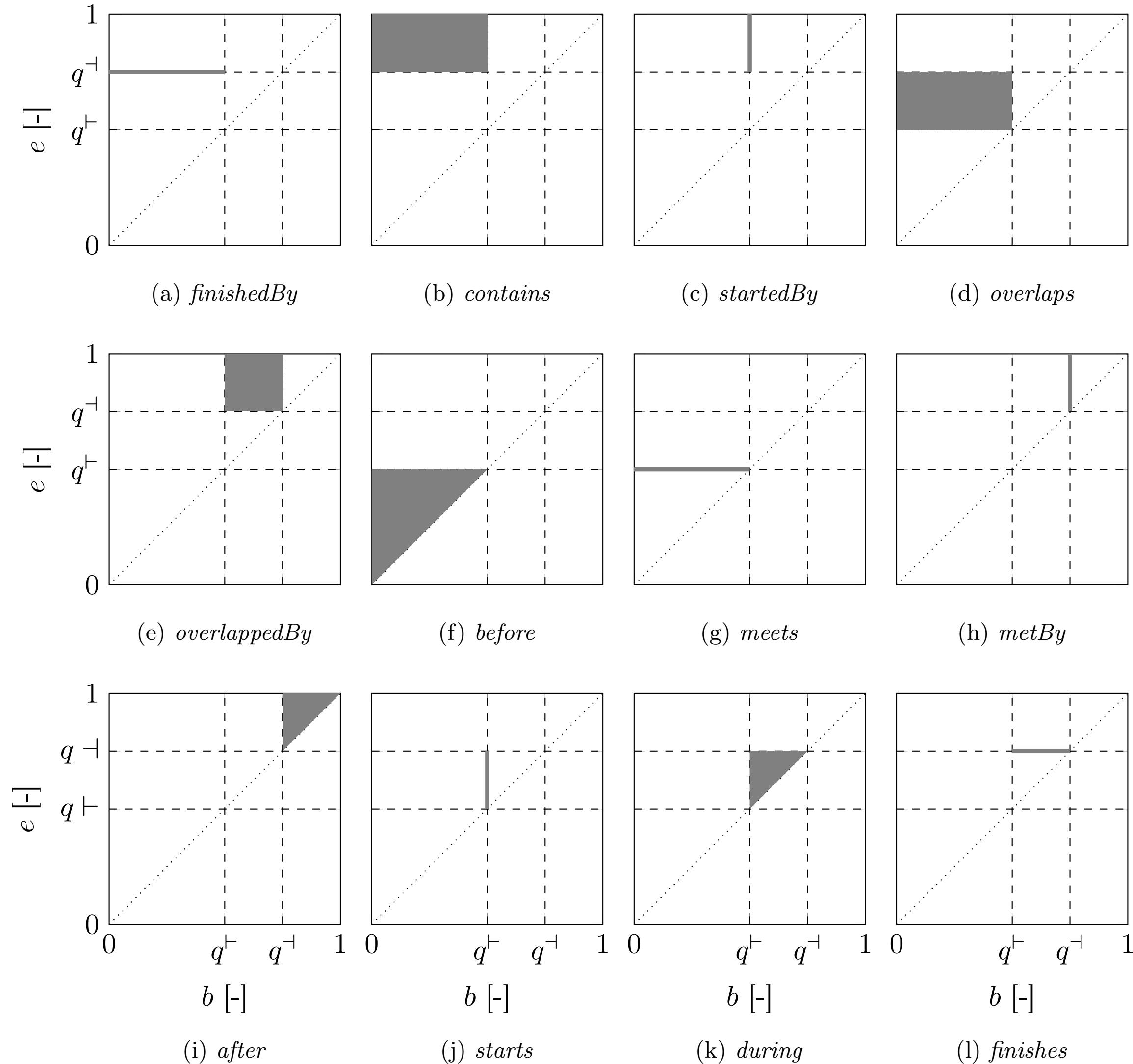


Figure 3.4: 12 queries on intervals. The gray area shows which data is affected by an interval query $(q^{\vdash}, q^{\dashv})$ when converted into a two-dimensional region query. The x-axis denotes the value of the interval's begin, the y-axis denotes the value of the interval's end. Note that no point lies beneath the diagonal and all points are lying inside $[0, 1]$ in every dimension. The *equals* query is a point query on $(q^{\vdash}, q^{\vdash})$ and $(q^{\dashv}, q^{\dashv})$ respectively (according to He et al. (2013) [97]).

A two dimensional region query always queries those objects which are contained in the space between the lower left $(b^{\vdash}, e^{\vdash})$ and upper right points $(b^{\dashv}, e^{\dashv})$ of the query rectangle including the borders. For instance, a converted *finishedBy* query would query the objects between $(0, q^{\dashv})$ and $(q^{\vdash}, q^{\dashv})$, whereas a converted *before* query would query the objects between $(0, 0)$ and $(q^{\vdash}, q^{\dashv})$.

3.3.3 (4) Adapt to *now*-relative Data

Before performing the query, the STPA also needs to incorporate *now*-relative data into the generated range queries for the following two cases:

1. The original n-dimensional interval query itself contains *now*-relative values, thus the end value in at least one dimension equals *now*.

2. The interval query matches the current value of *now* which is denoted as now^*.

For every one-dimensional interval query $(q^{\vdash}, q^{\dashv})$ the STPA has to determine how to change the converted two-dimensional region query. Table 3.2 lists how to adapt the resulting region query for each of the query types from Figure 3.4. Note that the *now*-relative values have been stored as the center value of the pyramids in the specific dimension d (c_d).

Despite the fact that the *finishes*, *finishedBy*, *starts*, and *startedBy* queries are not affected by any of the original eight relationships between objects as shown in Table 3.1, their conversion is shown in Figure 3.4 and Table 3.2 because one may want to define more specific temporal queries. In more detail, the steps required for adaption are described and proven in the following paragraphs.

before The *before* query is not affected by a *now*-relative query interval as only those rectangles are queried which completely lie before the actual query interval. A *before* query matches now^* if the end of the query interval is greater or equals now^*. Then the region query on the end dimension has to be extended to $\max(q^{\vdash}, c_d)$.

after The *after* query accepts all intervals beginning after the query interval has ended. If the query interval ends with *now*, the begin of the region query needs to be extended to $\min(c_d, now^*)$ on both dimensions, adjusting the begin of the query interval. Likewise, an *after* query matches now^* if the begin of the query interval is less or equals now^*. In that case, the begin on both dimensions of the region query has to be extended to $\min(q^{\dashv}, c_d)$.

overlaps The *overlaps* query accepts all intervals whose ends lie inside the query interval. If the query interval ends with *now* it has to be extended to $\max(c_d, now^*)$ to include those rectangles which are *now*-relative. An *overlaps* query matches now^* if now^* lies within the query rectangle. In that case, the range on the end dimension e has to be extended to $(\min(q^{\vdash}, c_d), \max(q^{\dashv}, c_d))$.

overlappedBy In contrast to *overlaps*, *overlappedBy* accepts those intervals, whose beginnings lie inside the query rectangle. If the query interval ends with *now*, the begin in the end dimension has to be extended, such that *now* is incorporated for the case that $now^* > c_d$. An *overlappedBy* query matches now^* if now^* lies within the query rectangle. Then, the begin on the end dimension of the region query has to be extended analogously to $\min(q^{\dashv}, c_d)$.

Table 3.2: Incorporating *now*-relative values into range queries. The original one-dimensional query $(q^{\vdash}, q^{\dashv})$ is mapped into a two-dimensional region query $((b^{\vdash}, b^{\dashv}), (e^{\vdash}, e^{\dashv}))$ with dimension b containing the begin values and dimension e containing the end values. The query has to be altered if $q^{\dashv} = now$ or if the query matches the current value of now. The value c_d is the value of the center point of the pyramids in dimension d of the converted interval and now^* is the current value of now.

querytype	$q^{\dashv} = now$	query matches now^*
before	not affected	$e^{\dashv} = MAX(q^{\vdash}, c_d)$
after	$e^{\vdash} = b^{\vdash} = MIN(c_d, now^*)$	$e^{\vdash} = b^{\vdash} = MIN(q^{\dashv}, c_d)$
overlaps	$e^{\dashv} = MAX(c_d, now^*)$	$e^{\vdash} = MIN(q^{\vdash}, c_d)$ $e^{\dashv} = MAX(q^{\dashv}, c_d)$
overlappedBy	$e^{\vdash} = MIN(c_d, now^*)$	$e^{\vdash} = MIN(q^{\dashv}, c_d)$
during	$b^{\dashv} = e^{\dashv} = MAX(c_d, now^*)$	$e^{\vdash} = MIN(q^{\vdash}, c_d)$ $e^{\dashv} = MAX(q^{\dashv}, c_d)$
contains	$e^{\vdash} = MIN(c_d, now^*)$	$e^{\vdash} = MIN(q^{\dashv}, c_d)$
starts	$e^{\dashv} = MAX(c_d, now^*)$	$e^{\vdash} = MIN(q^{\vdash}, c_d)$ $e^{\dashv} = MAX(q^{\dashv}, c_d)$
startedBy	$e^{\vdash} = MIN(c_d, now^*)$	$e^{\vdash} = MAX(q^{\vdash}, c_d)$ $e^{\dashv} = MIN(q^{\dashv}, c_d)$
meets	not affected	$e^{\vdash} = MIN(q^{\vdash}, c_d)$ $e^{\dashv} = MAX(q^{\vdash}, c_d)$
metBy	$b^{\vdash} = e^{\vdash} = MIN(c_d, now^*)$ $b^{\dashv} = MAX(c_d, now^*)$	$b^{\vdash} = MIN(q^{\dashv}, c_d)$ $b^{\dashv} = MAX(q^{\dashv}, c_d)$
finishes	$b^{\vdash} = e^{\vdash} = MIN(c_d, now^*)$ $b^{\dashv} = e^{\dashv} = MAX(c_d, now^*)$	$e^{\vdash} = MIN(q^{\dashv}, c_d)$ $b^{\dashv} = e^{\dashv} = MAX(q^{\dashv}, c_d)$
finishedBy	$e^{\vdash} = MIN(c_d, now^*)$ $e^{\dashv} = MAX(c_d, now^*)$	$e^{\vdash} = MIN(q^{\dashv}, c_d)$ $e^{\dashv} = MAX(q^{\dashv}, c_d)$
equals	$e^{\vdash} = MIN(c_d, now^*)$ $e^{\dashv} = MAX(c_d, now^*)$	$e^{\vdash} = MIN(q^{\dashv}, c_d)$ $e^{\dashv} = MAX(q^{\dashv}, c_d)$

during The *during* query accepts all intervals which completely lie inside the query interval. A *now*-relative query interval would then be required to extend the end of the region query in both dimensions to $\max(c_d, now^*)$. A *during* query matches now^* if now^* lies within the query rectangle. Therefore, the range on the end dimension e has to be extended to $(\min(q^{\vdash}, c_d), \max(q^{\dashv}, c_d))$.

contains The *contains* query accepts all intervals which contain the query interval. Therefore, the region query must cover the region at which the begin is $\leq q^{\vdash}$ and the end is $\geq q^{\dashv}$. If the query interval is *now*-relative, the region query has to be extended along the end dimension with $e^{\vdash} = \min(c_d, now^*)$, such that those intervals are queried whose end value is *now*. A *contains* query matches now^* if now^* is greater than the query interval's end. As $e^{\dashv}$ already is set to 1, only $e^{\vdash}$ has to be set to $\min(q^{\dashv}, c_d)$.

starts The *starts* query accepts all intervals whose begin is equal to the begin of the query interval and whose end lies within the query interval. A *now*-relative query interval imposes the extension along the end dimension, such that those intervals are queried which start with the query intervals start but end with *now*, resulting in $e^{\dashv} = \max(c_d, now^*)$. The *starts* query matches now^* if now^* lies within the query interval, which results to an extension of the range query along the end dimension to $(\min(q^{\vdash}, c_d), \max(q^{\dashv}, c_d))$.

startedBy The *startedBy* query accepts all intervals whose begin is equal to the begin of the query interval and whose end lies outside the query interval. As the *now*-relative intervals have to be excluded from the result set if the query interval is *now*-relative, the region query has to be reduced by $e^{\vdash} = \min(c_d, now^*)$. A *startedBy* query matches now^* if the end of the query interval is $\leq now^*$. Then the region query has to be reduced along the end dimension to $(\max(q^{\vdash}, c_d), \min(q^{\dashv}, c_d))$.

meets The *meets* query accepts all intervals which end with the begin of the query interval. Therefore, the *meets* query is not affected by a *now*-relative query interval. It matches now^* if the begin of the query interval equals now^*. If so, the range query has to be extended to $(\min(q^{\vdash}, c_d), \max(q^{\vdash}, c_d))$ along the end dimension. Note that the *meets* query does not incorporate $q^{\dashv}$.

metBy The *metBy* query accepts all intervals which start with the end of the query interval. If the query interval is *now*-relative, not only the begin of the region query has to be extended to $\min(c_d, now^*)$ but also the end along the begin dimension $b^{\dashv}$ needs to be extended to $\max(c_d, now)$ such that those intervals are queried which have a begin that matches now^*. A *metBy* query matches now^* if $q^{\dashv} = now^*$. Then the region query has to be extended along the begin dimension to $(\min(q^{\dashv}, c_d), \max(q^{\dashv}, c_d))$. Note that the *metBy* query does not incorporate $q^{\vdash}$.

finishes The *finishes* query accepts all intervals which have the same end as the query interval and whose begin lies within the query interval. Given a *now*-relative query interval, the region query has to be extended along both dimensions to $(\min(c_d, now^*), \max(c_d, now^*))$. A *finishes* query matches now^* if $q^{\dashv} = now^*$. In order to query those intervals which are *now*-relative, the region query has to be extended along the end dimension to $(\min(q^{\dashv}, c_d), \max(q^{\dashv}, c_d))$. Note that the *finishes* query does not incorporate $q^{\vdash}$. In addition, the query needs to be extended along the begin dimension to shift the original query interval to now^* by $b^{\dashv} = \max(q^{\dashv}, c_d)$.

finishedBy The *finishedBy* query accepts all intervals which have the same end as the query interval and whose begin lies outside the query interval. As the region query already is 0 at the begin dimension, it has only to be extended along the end dimension to $(\min(c_d, now^*), \max(c_d, now^*))$ if $q^{\dashv} = now$. Likewise, if $q^{\dashv} = now^*$, the region has to be extended along the end dimension to $(\min(q^{\dashv}, c_d), \max(q^{\dashv}, c_d))$. Note that the *finishedBy* query does not incorporate $q^{\vdash}$.

equals The *equals* query accepts all intervals which have the same begin and the same end as the query interval. If the query interval is *now*-relative, the region query has to

be extended along the end dimension to $(\min(c_d, now^*), \max(c_d, now^*))$. An *equals* query matches now^* if $q^{\dashv} = now^*$. Then the query has to be extended along the end dimension to $(\min(q^{\dashv}, c_d), \max(q^{\dashv}, c_d))$

3.3.4 (5 + 6) Query the Pyramid Space

Querying the pyramid space contains two steps: First, the d two-dimensional region queries have to be merged into one $2d$-dimensional query in the pyramid space. This $2d$-dimensional query is then split up into $4d$ range queries on the underlying B$^+$-tree as suggested for the original Pyramid technique [47].

3.3.5 Query Alternatives

The described query alternative usually extends the queried space in order to incorporate *now*-relative data. This causes many false positive results which have to be removed from the result set by matching the results to the original query. In order to minimize the number of false positives, two query alternatives are described in this section.

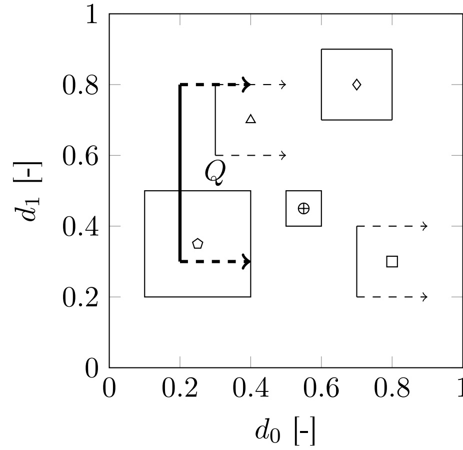
In the first alternative, whenever the stored *now*-relative data has to be accessed, the query intervals are not expanded but in addition to the spatial-query only those *now*-relative elements that lie on the median of the data set are queried which are affected by the query. Both result sets are then combined with a set operator.

In the second alternative, the *now*-relative elements are stored in separate indices. As bi-temporal data is investigated, four indices have to be queried in that alternative. One which only contains *now*-relative valid-time data, one which only contains *now*-relative transaction-time data, one which only contains elements which are *now*-relative in both temporal dimensions and one which does not contain any *now*-relative data. Therefore, every query is distributed into up to four queries on the separated indices and the results combined by set operators. As the elements in the underlying B$^+$-tree are ordered, the query results could be joined in linear time in both alternatives. Unfortunately, the costs for the additional queries exceed the benefits of much less false positive results. The evaluation of these indices with the workloads described in Section 3.5.1 shows that the alternatives perform worse by up to three orders of magnitude. Therefore, they are not investigated any further.

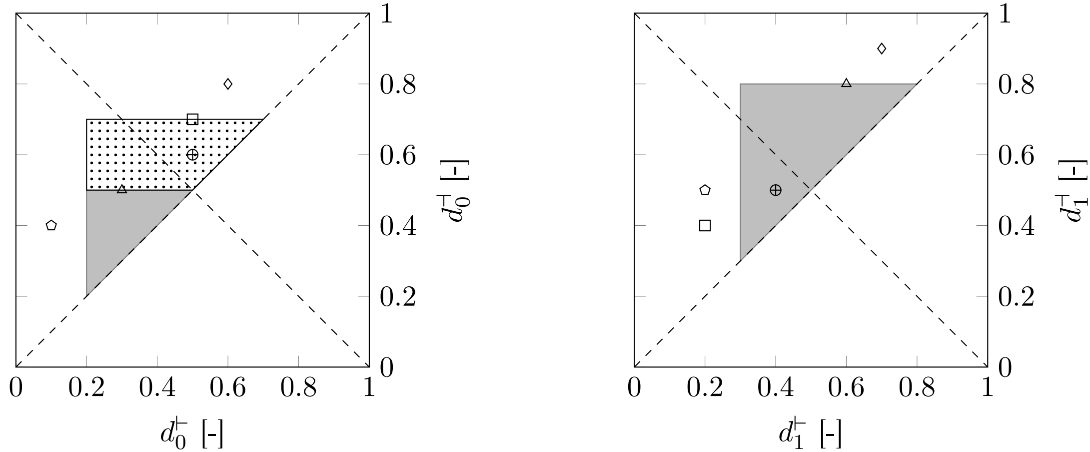
3.4 Implementation of the STPA

The STPA is implemented with Java 8. Figure 3.6 gives an overview of the implemented class structure which is described in more detail in this section.

It may be divided into a storage layer (below) and the index layer (above) which are divided by the dashed line. The storage layer manages the storage of any object which implements the `Storable` interface. Using reflection, one may implement a general wrapper which is able to generate a `Storable` representation of any object type. For this thesis, only concrete `Storable` implementations are used, each created for the types which need to be stored throughout this thesis. The actual `StorageManager` is abstract and the two concrete implementations are an on-disk and an in-memory manager. In order to count the number of accesses on a `Storable`, a chain of `StorageManager` instances is implemented, with the actual manager on the inner level, wrapped by a counting `StorageManager` on the outer level. In addition, buffers may be wrapped around a `StorageManager` instance.



(a) The original query rectangle is $Q = (0.2, 0.3), (now, 0.8)$. The query accepts all rectangles which completely lie within the query rectangle. Note that *now* is depicted as arrow with a length of 0.2. In order to validate the existence of □, *now* should at least be 0.7, the begin value of □ in d_0.



(b) Following Table 3.1, the given *contained* query is converted into two *during* queries. These during queries are mapped to region queries according to Figure 3.4. According to Table 3.2, the region query for d_0 is extended along dimension $d_0^{\dashv}$ to 0.7 because $q_0^{\dashv} = now$ and $now^* = 0.7 > 0.5 = c_0$, the median of the data set in dimension d_0.

$d_0^{\vdash}$	(0.2, 0.7)	$d_0^{\dashv}$	(0.2, 0.7)
$d_1^{\vdash}$	(0.3, 0.8)	$d_1^{\dashv}$	(0.3, 0.8)

(c) The merged region query basically consists of four range queries (dimension/range), one along each dimension. This query has to be converted by the Pyramid Technique to query the underlying B^+-tree. The candidates for the result set are those rectangles which are contained in both region queries.

Figure 3.5: Example of a *contained* [78] query q in a two dimensional space with one temporal (d_0) and one spatial (d_1) dimension and its conversion by the STPA with $now^* = 0.7$.

The index layer contains the implementations of the index structures and the query system. The basic structure is the `NDRectangleKey` which consists of a MBR, represented by a

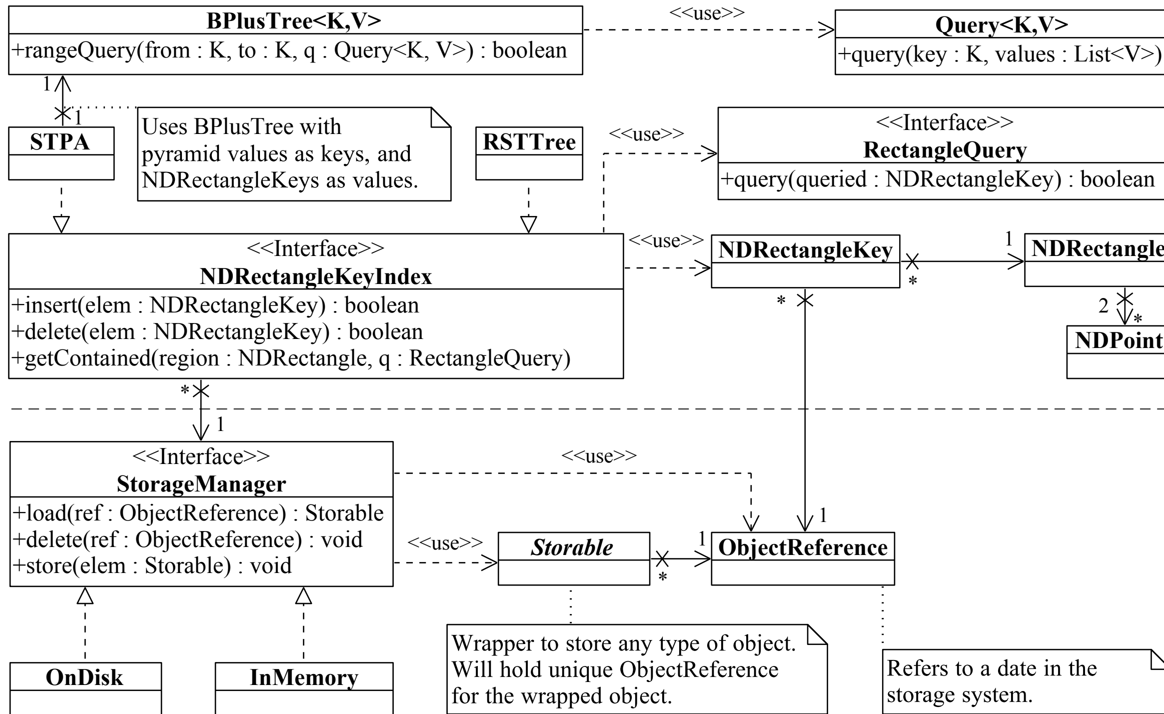


Figure 3.6: UML class diagram of the STPA. The upper part shows the query system and indexing methods, the lower part shows the structure of the underlying storage manager. Any type of object may be stored, if it has been wrapped into a `Storable`. Every *Storable* is identified by a unique `ObjectReference`.

`NDRectangle` and an `ObjectReference` which is the reference on the `Storable` that represents the object. A `NDRectangle` represents a n-dimensional (hyper-) rectangle. It consists of two `NDPoints`, one for the begin and the other for the end values in every dimension. If the end value equals a `NOW`-constant, it is considered to be *now*-relative. Any index is an implementation of the `NDRectangleKeyIndex` interface. It will be used as basis for the comparison of index structures in this chapter. In Chapter 6, a more general definition of an index interface is given, because not all third-party index implementations incorporated in this thesis provide something comparable to a `Storable` or `StorageManager`.

The general procedure of a query is visualized in Figure 3.7. Berchtold et al. (1998) [47] suggest to store all query results in a point set. Every element in this point set is then matched to the original query. In order to keep the necessary additional space at a minimum, the STPA uses the Visitor pattern [85].

In order to query the STPA, the client has to provide a query region and a query instance. The query instance is a visitor and the STPA a visitable. Thus, for every element accepted by the query, the given query instance is invoked by the STPA. The STPA converts the region query into $4d$ range queries and performs them on the underlying B^+-tree [67]. It provides a begin and an end value for the range query and a query object. Every time the B^+-tree finds a match for the given range query, it invokes the given query object. The query object then matches the result from the range query against the original query. As the different range

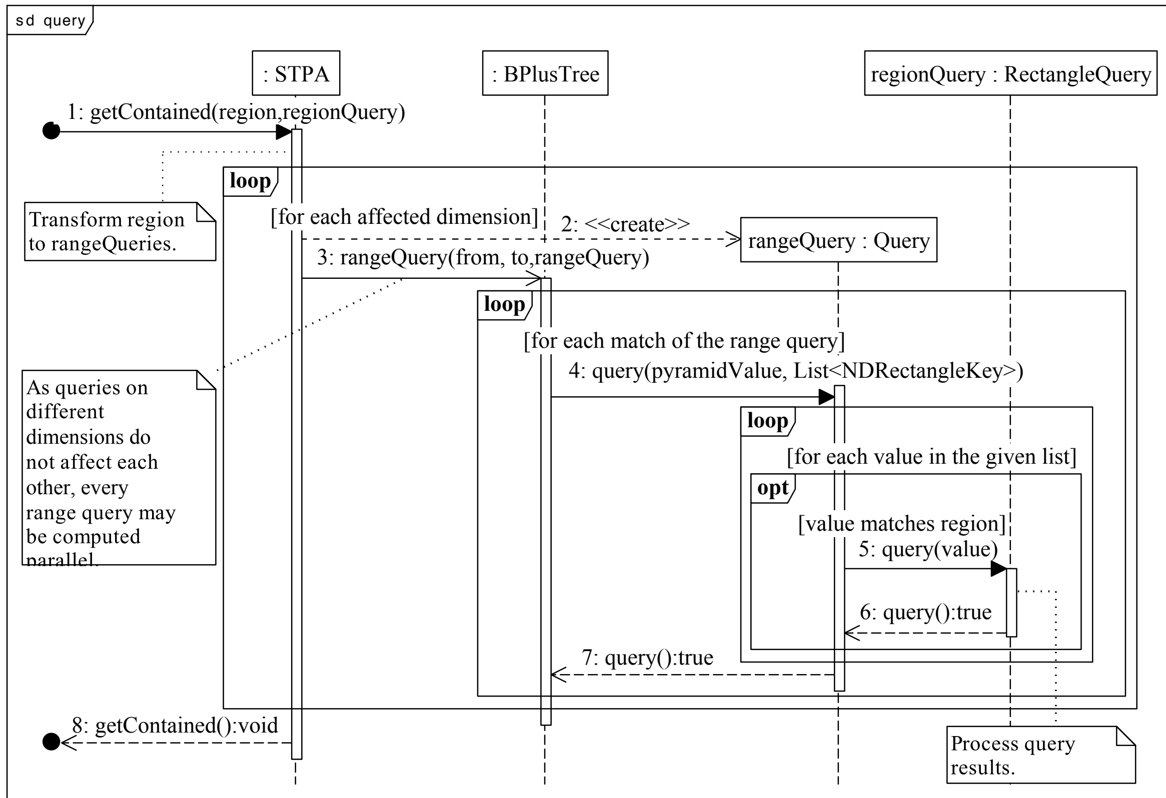


Figure 3.7: UML sequence diagram of the STPAs parallel query technique. It is assumed that the original d-dimensional query is converted into $4d$ range queries as described in section 3.3. The structure of the classes used in this diagram is shown in Figure 3.6.

queries on the B$^+$-tree are independent to each other, the underlying range queries in the STPA may be computed in parallel.

3.5 Classic Workload Evaluation

This section evaluates the query performance of the STPA using a workload generator derived from Saltenis and Jensen (1999) [172]. For comparison, an own implementation of the RST-tree is also evaluated with this workload generator. To the knowledge of the author, the RST-tree is the only competitor which is designed to index discretely changing spatio-temporal data instead of moving objects. The RST-tree is implemented with the same Visitor-based query technique as the STPA (Section 3.4). Not only a constant but also the dynamic time horizon [173] is used (Section 2.2.2) for time parametrization.

3.5.1 Workload Generator and Setup

One workload is generated as follows. An index is initialized with `initialSize` elements whose end values on the transaction-time axis are set to *now*. Each of these elements with a *now*-relative end value is considered to be part of an *active* history. A history is a chain of

versions of one and the same object which changes its appearance. Within every evaluation step `incSize` elements are inserted, in which a ratio of `startPercentage` elements start a new active history. A ratio of `endPercentage` elements end a still active history by updating the last element in that history with an element whose end value in transaction-time is now^*. A ratio of `updatePercentage` elements continue an active history by adding an element to a history whose end value is *now* on the transaction-time axis. Adding an element to a history always means that the end value on the transaction-time axis of the latest element is updated to now^*. Therefore, the start value on the transaction-time axis of the newly inserted element is set to now^*. The bi-temporal history of two objects A and B is exemplified in Figure 3.8 with $now^* = 0.7$. The history of A consists of three transactions, each changing the valid-time interval of the object. As the last element in transaction time has an end value of *now*, the history is *active*. History B only consists of one transaction and is *inactive*.

Table 3.3: Setup parameters for the in-memory and on-disk evaluation plus uniformly distributed, clustered, and skewed data sets.

parameter	value(s)	
`startPercentage`	0.1	
`endPercentage`	0.1	
`updatePercentage`	0.8	
`distribution` `validTimeDistribution`	uniform	$in[0,1]$
	normal	$stdDev = 2\sqrt{log_e(2)}$ $mean = 0.25$
	skewed normal	$stdDev = 0.3$ $mean = 0.25$ $skew = -1.0$
`vtInfinityProbability`	0.1	
`maxValidTimeLength`	0.1	
`maxElementSize`	0.1	
`initialSize`	1000	
`incSize`	11000	
`queries`	1000	
`querySize`	0.2	
`dimensions`	$5, 10, 15, 20, 30, 40, 50$	
`StorageManager`	`InMemory`	
	`OnDisk`	4096 bytes per block 0,50 blocks in buffer

The distribution of the elements in transaction-time is given by the filling process described above. The distribution along the valid-time axis is given by a `validTimeDistribution` which may either be a uniform, gaussian or skewed normal distribution. Although it is possible to vary the range and concrete behavior of the distributions, the evaluation only uses data sets which are generated with one of the following settings:

- A uniform distribution in $[0,1]^d$.
- A gaussian distribution with a standard deviation of $2\sqrt{log_e(2)}$ and a mean value of 0.25 in order to create data clusters at 0.25 and most elements lying in $[0,1]^d$.

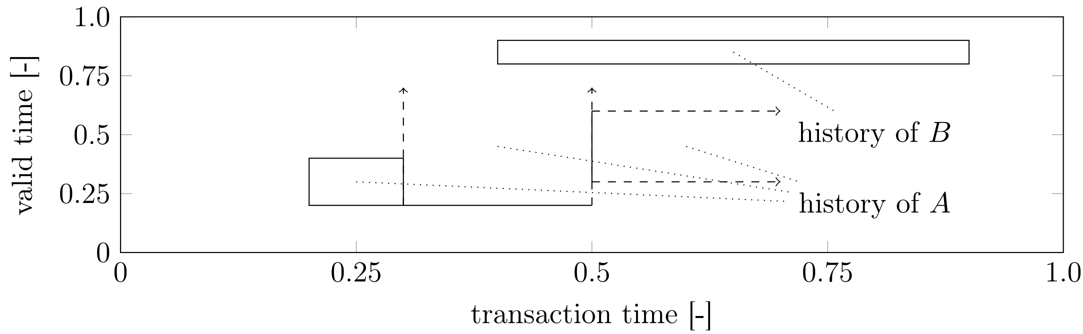


Figure 3.8: Exemplified history of two objects A and B in the bi-temporal space.

- A skewed gaussian distribution with a standard deviation of 0.3, a mean value of 0.25, and a skew of -1 in order to achieve a maximum skewness. The skewness is applied on a random number y by $y = (1 - e^{-skew*y})/skew$.

In addition, elements which lie outside of $[0, 1]$ in any dimension are discarded. Example point clouds of these distributions are visualized in Figure 2.8 on page 21.

To create *now*-relative valid-time intervals, the parameter `vtInfinityProbability` denotes the likelihood for setting the end time on the valid-time axis of a newly created element to *now*. For all non-*now*-relative elements the parameter `maxValidTimeLength` denotes the maximum length of the valid-time interval. The length is always uniformly distributed in [0, `maxValidTimeLength`]. The begin and end values in all non-temporal dimensions are also created with either uniform, gaussian, or skewed gaussian distributions. All non-temporal values are created with the same distribution `distribution` and the maximum length of the intervals is given by `maxElementSize`. The number of dimensions is adjustable but all elements have a transaction- and a valid-time dimension at least. The dimensions are chosen with respect to the long computation time for one workload and in order to show the general behavior of the structures with increasing dimensionality. As the impact of the curse of dimensionality may be expected between 5 and 15 dimensions, smaller numbers of dimensions are chosen for evaluation. After every insertion of `incSize` elements, the structure is queried by an amount of `queries` in $[0, 1]^d$ uniformly distributed *contained* queries. The maximum length of the queries in every dimension is given by `querySize`. This is possible because the valid and transaction-time values are also lying in $[0, 1]$. now^* starts at 0.25 in order to model that some time already has passed and is increased by a constant for every insert or update operation on the evaluated index. This constant is defined such that now^* is 0.75 at the end of the workload generation. Different block and buffer sizes are evaluated but the evaluation is concentrated on a block size of 4096 byte and a buffer size of 0 due to the fact that a greater buffer size only shifts the results but does not change the general conclusion if the number of inserted elements is big enough. In order to keep the number of elements in one workload at a minimum without falsifying the outcome, the impact of different buffer sizes is not discussed in detail.

Keep in mind that the block size of the evaluated indices is only crucial for the on-disk case, since the maximum number of entries in one node is constant for the in-memory case. For each workload setting, the node sizes of the indices have been optimized. For the implementation

of the R^{ST}-tree, the best maximum node size is 32 entries and the best maximum node size for the underlying B^+-tree of the STPA is 40 entries.

The STPA can only be improved further by altering the center of the pyramids. The center of the pyramids is set to the median of the data set at the beginning of the evaluation which is almost optimal. Since the temporal values change over time and therefore the approximated median changes over time, the Pyramid Technique, whose center is set to a constant, only provides approximately optimal query results. As in the original paper [47], experiments show that the benefits in the query cost of providing an optimal center of the pyramids is outweighed by the additional rebuilding cost. Contrary, when using an approximately optimal median, the additional rebuilding cost is negligible [47]. The effect of a non-optimized median is investigated in Section 3.5.2.

The R^{ST}-tree can be improved either by using a constant or a dynamic parametrization value and the parameter α. For the workloads described in this section, both types of parametrization values are compared and the constant parametrization shows the best results. Note that the α and optionally α_W - parameter is adapted anew for each evaluation setup and especially for each spatio-temporal distribution. As a reminder, the α-value denotes if the bi-temporal ($\alpha > 0$) or spatial part ($\alpha \leq 0$) of the MBRs should be prioritized when computing their margin, overlap and volume. The α_W parameter is a control constant for the expected window querying length when the R^{ST}-tree uses the time horizon for dynamic parametrization. For the in-memory case, the node size in the R^{ST}-tree can be improved.

The setup of the workload generator is listed in Table 3.3. 100 workloads are generated for every setup and both, an in-memory and an on-disk working `StorageManager`. Every workload contains 10 evaluation steps with an initial size of 1000 elements. Every evaluated index contains 100000 elements at the end of one workload.

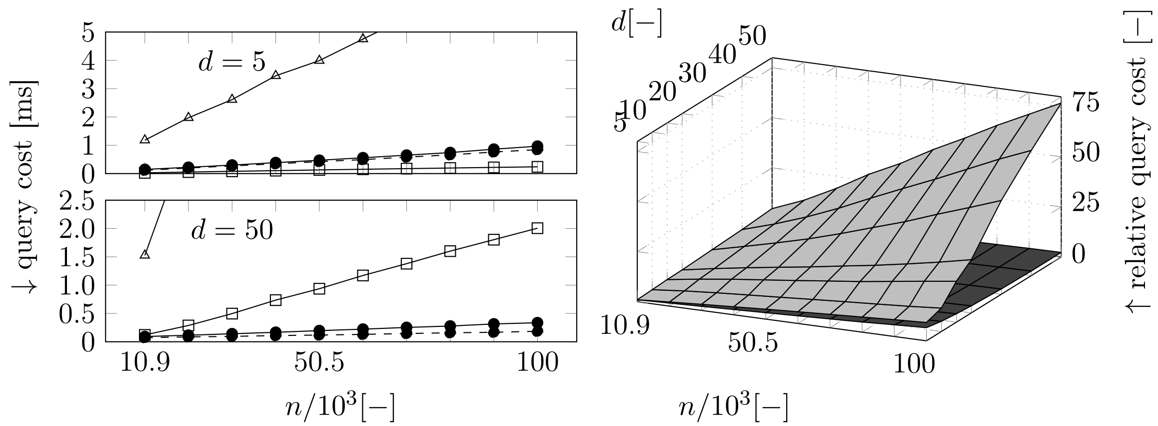
In order to provide a reproducible evaluation setup, every pseudo random number generated within one workload depends on a certain random seed. Using this seed, one may reproduce every number as it was generated in a previous generation of a workload with the same set of parameters.

3.5.2 Results

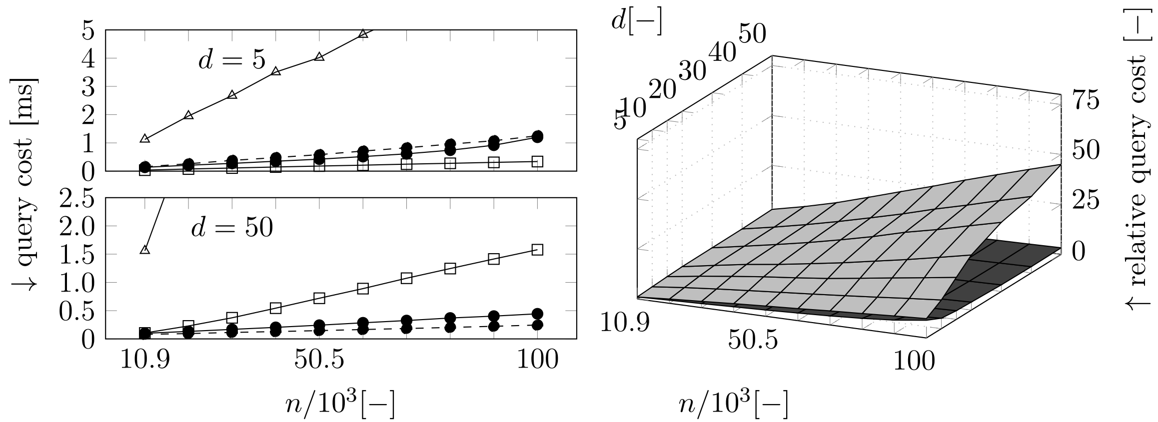
The workloads described in the previous section are computed and the results are presented in this section. Each time the queries in a workload are computed, their performance is measured. For the in-memory case, the CPU-time is crucial since both indexing methods do not need much additional memory storage. Note that the maximum number of entries in the nodes of both structures remains constant for the in-memory case, whereas the node size and not the number of entries remains constant for the on-disk case. The number of I/O operations is the most relevant value for the on-disk case. Therefore, for the in-memory case, the CPU time for each query is measured. The measurement of the CPU time in Java is discussed in detail in section 5.4. As a single workload contains over $1.1 \cdot 10^6$ invocations of the basic methods, the impact of the JIT compilation is negligible for the measurement of the CPU time. For the on-disk case, the number of I/O operations is measured using a certain `StorageManager` which counts each load and store operation.

For every evaluation setup three diagrams are provided:

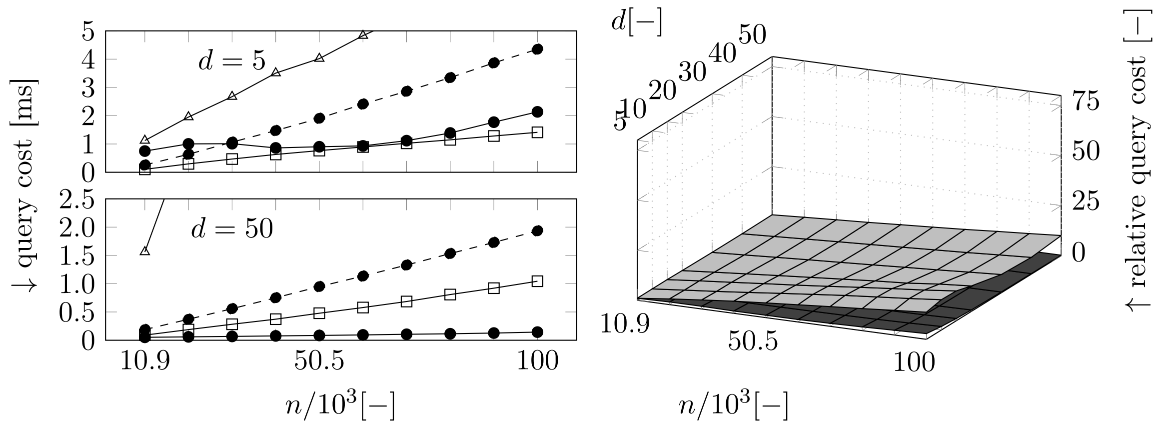
- A three-dimensional diagram showing the means for every data point (n, d) with n being the number of elements in the index and d being the number of dimensions. The



(a) uniform distributions

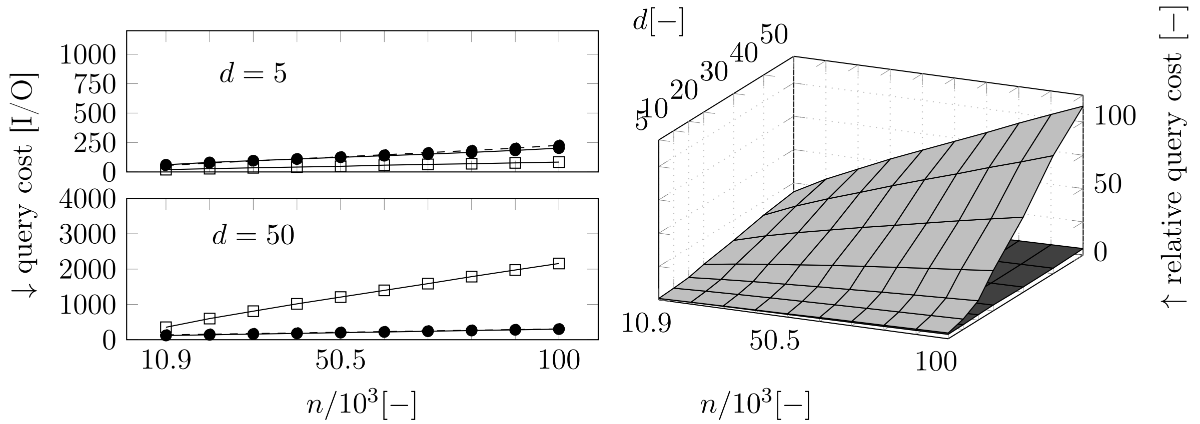


(b) gaussian distributions

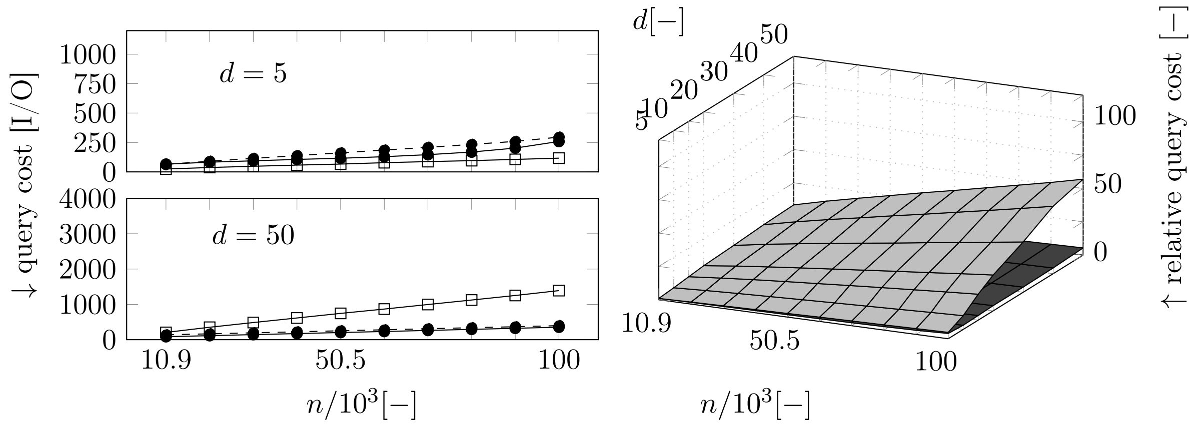


(c) skewed gaussian distributions

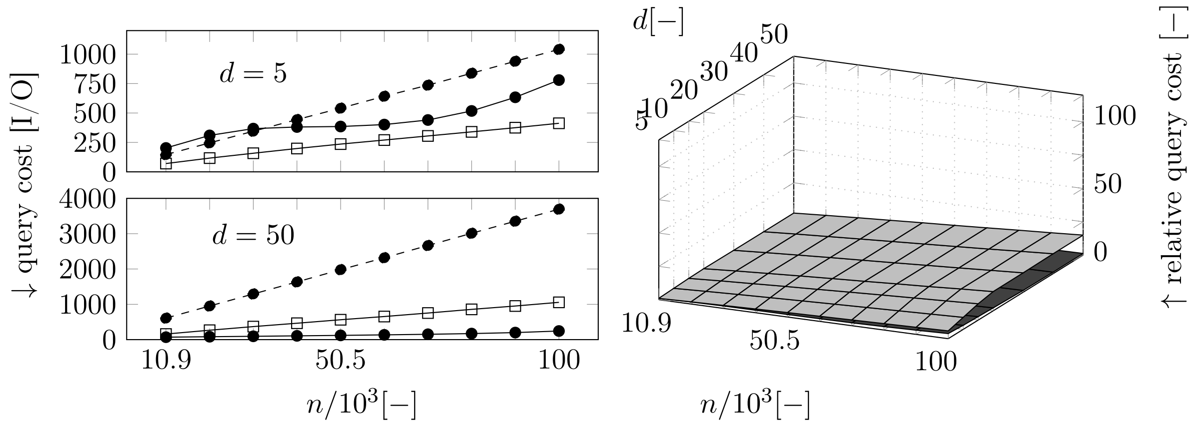
Figure 3.9: Comparison of the STPA (•, dark gray) with approximated optimal median (black), default median (dashed) and the R^{ST}-tree (□, light gray) in the in-memory case. The absolute query cost of the sequential scan (△) is not shown fully to achieve a better comparability between the other methods.



(a) uniform distributions



(b) gaussian distributions



(c) skewed gaussian distributions

Figure 3.10: Comparison of the STPA (●, dark gray) with approximated optimal median (black), default median (dashed) and the R^{ST}-tree (□, light gray) in the on-disk case. The query cost of the sequential scan always equals the number of elements.

mean at every data point is divided by the mean of the first data point $(10900, 5)$. This improves the visibility of the relative growth of the query cost with respect to the increasing dimensionality. For an improved visibility, only the data from the STPA with an approximately optimized median and the R^{ST}-tree is shown.

- Two two-dimensional diagrams showing a Line Plot of the absolute mean values for 5 and 50 dimensions.

For both, the in-memory (Figure 3.9) and the on-disk (Figure 3.10) case, the R^{ST}-tree has a better query performance than the STPA for lower dimensions but is clearly outperformed by the STPA for higher dimensions. Taking the trend for an increasing number of elements into account, the STPA is more efficient in higher dimensions.

The better query performance of the STPA in the in-memory case is likely to result from the fact that for an increasing number of dimensions by a constant maximum number of entries in every node the R^{ST}-tree has to perform an increasing number of floating-point computations when computing the overlaps, margins and volumes of the nodes MBRs. For the on-disk case, the performance of the R^{ST}-tree is reduced in higher dimensions since the maximum number of entries decreases. That is why more nodes have to be created for storing the same number of elements and therefore more nodes have to be accessed when executing a query. Neither the maximum number of entries in the dir-nodes nor the size of the dir-nodes of the underlying B^+-tree of the STPA is affected by an increasing number of dimensions because every entry is always identified by a one-dimensional value. However, the maximum number of entries in the leaf-nodes of the underlying B^+-tree is affected by an increasing number of dimensions in the on-disk case and the STPA also has to perform more sub-queries on high-dimensional data which causes a slightly worse query behavior in the in-memory case.

For different distributions of data the STPA and the R^{ST}-tree show the same general behavior but especially the impact of the median in the STPA varies. For uniformly and gaussian distributed data choosing the default median of 0.5 seems to have less impact on the evaluation results as the STPA with the default median is only slightly worse than the STPA with approximately optimized median. For the in-memory case (Figure 3.9), the mean values seem to be better for the configuration with the default median but as the results for optimized and default median also vary about 0.4 a general conclusion on the impact of the median cannot be made. Such an impact can be seen for the skewed distribution. There, the approximately optimized median shows better results than the default median setting. The dent in the plot for the skewed gaussian distribution in lower dimensions (Figures 3.10c, 3.9c) may be caused by the greater impact of both time dimensions. Especially if it is recalled that the time moves from 0.25 to 0.75 through every single evaluation.

With respect to the 3D-plots which show the relative means of the STPA with approximately optimized medians and the R^{ST}-tree, the relative difference is larger for uniformly, less for gaussian and even lesser for skewed gaussian distributed data. This means that the increase of the query cost in the R^{ST}-tree depends on the distribution of the data, whereas the STPA seems to have a nearly equal increase for all three distribution types.

A similar behavior for the costs of insertion, update, and deletion in the indexing methods can be observed. In difference to the query cost, the STPA outperforms the R^{ST}-tree also in absolute terms and lower dimensions.

3.6 Conclusions

The evaluation shows that the STPA clearly outperforms the R^{ST}-tree in terms of query performance and increasing dimensionality. Beside that, the whole setup is based on assumptions and best guesses. It would take a much greater and more exhaustive workload generation to prove the conclusions made in this section and it is to be questioned if a proof is possible with that large number of impact factors. As a reminder, not only the number of elements but also their size, distribution and the order of their insertion, deletion and update influence the performance behavior. In addition, the size and distribution of the query rectangles influences their performance. Lastly, one has to recall the impact of the ongoing time. A query performs different when executed at different times.

Nonetheless, the evaluation does support the main thesis of this chapter, which is the STPA being the currently best solution for indexing discretely changing high-dimensional spatio-temporal data. But the evaluation is not suited to unfold those settings which do cause bad performance behavior. Therefore, one needs either to explore the complete parameter space, or to analyze the structure of the used methods to unfold their general behavior, including best and worst case scenarios. The latter is done in Chapter 5.

Aside the further discussion on the workload generator, one should keep in mind that the STPA is based on the B^+-tree and therefore may easily be implemented on top of many existing databases. In addition, the query system supports the parallel computation of the sub-queries (see Section 3.4) which should decrease the total system time a query is running.

Chapter 4

Automated Performance Comparison

This chapter introduces a new approach for the automated comparison of the performance of complex data structures which is called Interface Based Performance Comparison (IBPC) technique. A concrete implementation, the Performance Test Automation Framework (PTAF), is described in Chapter 5. Whereas the motivation for a new technique and a complete overview of the structure is given here, the detailed description of the implementation of every necessary module in PTAF is provided in Chapter 5 and the analysis and evaluation of PTAF is provided in Chapter 6. The general idea is partly published in [133].

In Section 3.5, two spatio-temporal indexing techniques are compared to each other with a focus on their query performance on datasets with a high number of dimensions (>10). It indicates that the Spatio-Temporal Pyramid Adapter (STPA) shows a much better query performance in higher dimensions than the R^{ST}-tree. Because of the high number of factors that influence or might influence the performance behavior of the structures, a general proof of the performance of the structures cannot be given. In addition, the used benchmarks may fulfill the requirements of a good benchmark (Section 2.3) but may not uncover all performance features inherent to the evaluated structures. That is, the workload generator reveals the impact of an increasing number of dimensions on the structures for several data distributions but may not be useful to describe the general average behavior for all possible data sets. When searching for the best performing structure regarding a certain functionality, e.g. the query performance in high-dimensional spaces, one should use a specialized benchmark. But if one wants to analyze and compare the general performance of several structures, e.g. the performance of insertion, deletion and the execution of queries, a new approach is needed. The usage of a benchmark may be biased in two different ways. First, a structure may be designed to perform well in one benchmark, e.g. executing queries on uniformly distributed data. Secondly, a benchmark may not cover a certain behavior of the analyzed structures. For instance, a spatio-temporal high-dimensional index may perform well on the data distributions provided by the benchmark but (unintentionally) perform much worse when using another, not tested, distribution. The difference is that in the first case, the structure exploits a speciality of the benchmark. In the second case, the benchmark is not correctly set up. That is, the structure relies on a benchmark that fully tests its capabilities in the second case and in the first case, the structure uses a weakness of the benchmark to be proven as efficient. The following section shows how the problems of possibly biased benchmarks are faced by the IBPC.

4.1 Overview and General Idea

The Interface Based Performance Comparison (IBPC) technique faces the problems of possibly biased benchmarks. Implementations like PTAF should produce performance tests that are unlikely biased and can be used to automatically compare a given set of competitors. The most likely source of biased benchmarks lies in the manual generation of the respective workloads or parameters of a workload generator. Therefore, the IBPC tries to minimize the user input and maximize the automation.

Although in [30] it is stated that the goal of covering the source code to create workloads may be a "a relatively naive way of designing workloads", it is the main concept in this thesis for the generation of workloads. All alternatives for the automatic generation of workloads presented in [30] and, to the knowledge of the author, elsewhere, somehow base on the structural code coverage or require more user input (see Section 2.5). As the user input is to be minimized to avoid biased workloads, the only logical choice for the generation of performance tests are test sets that are created with the goal of a maximized coverage. The measurement of coverage and the coverage type chosen for the workload generation is described in Section 5.1. The automated generation of workloads addresses the problem of possibly biased benchmarks and benchmarks which (unintentionally) miss certain configuration of the problem, e.g. the distribution of a spatio-temporal data set.

The performance of the automatically generated test sets must be measured with respect to the used programming language. For the use of Java, a new measurement technique is discussed in Section 5.4. The comparability of the performance measurements can be ensured in two ways. First, all competitors share the same interface. Doing so, the test sets of each competitor become interchangeable. Secondly, this offers the opportunity to execute the test sets of each competitor on each other competitor. In combination with the code coverage of each test set on each system under test (SUT), the performance measurement can be weighted with respect to the achieved coverage. By using the coverage as performance weight, the structural differences of the competitors are incorporated into their comparison. The usage of the coverage as a performance weight addresses the problem of structures which only perform well for a certain type of inputs, e.g. a certain benchmark.

The general idea behind the comparison procedure is presented in Algorithm 3. Given a set of n SUTs S_i, $i \in \{1, ..., n\}$, which all implement an interface I, the algorithm generates a test set T_i for each S_i. Each test set only consists of sequences of the invocations of functions and initializers with the respective parameters defined by I. All test sets T_j, $j \in \{1, ..., n\}$, are then executed on each S_i and the measured performance $p_{ij} \geq 1$ is stored. This way, the performance of a test set created on the basis of a certain SUT is measured when executed on all SUTs. The differences between those executions unfold the differences in the performance of the competitors. In order to measure differences in the performance tests, the coverage of the test sets T_j is measured for each S_i and stored as c_{ij}. This coverage is used in two ways. First, the differences in the coverage computed for test sets which are created for different SUTs may unfold functional differences between the corresponding SUTs. Secondly, for a fair performance comparison, not only the better performing SUTs are benefited, but also those which are covered by all test cases in an equal manner. Test sets with a high coverage on an insular approach, which are designed to perform well on a certain set of parameters, most likely do not have a high coverage on those SUTs which support good performances on the complete parameter space. With that in mind, the performance is weighted by the coverage. Note that a lower value is always considered to indicate a better performance.

Input : Interface I, SUTs S_i with default initializers
Output: combined coverage c_i and weighted performance p_i for each S_i

```
1  foreach S_i do
2  |  T_i = GenerateTestSet(I, S_i)
3  end
4  foreach S_i do
5  |  foreach T_j do
6  |  |  p_ij = ExecuteTestSet(S_i, T_j)
7  |  |  c_ij = ComputeCoverage(S_i, T_j)
8  |  end
9  |  cov_ij = c_ij / c_ii
10 end
11 c_i = prod_{j=1}^{n} (cov_ij)^(k/n)
12 p_i = sum_{j=1}^{n} p_ij / (cov_ij)^k / n
```

Algorithm 3: General approach of the IBPC technique.

In more detail, the coverage of T_j on S_i first has to be mapped into $[0, 1]$, as there might be unreachable code fragments which alter the impact of coverage weights. Therefore, $cov_{ij} = \frac{c_{ij}}{c_{ii}}$, with the assumption that a test set which has been created with a maximized coverage on S_i, has the maximum coverage on S_i. Given a test generation system which does not certainly create a test set with maximized coverage, one may also use

$$cov_{ij} = \begin{cases} \dfrac{c_{ij}}{max_i} & \text{if } c_{ij} < max_i \\ 1 & \text{otherwise} \end{cases} \tag{4.1}$$

with max_i being the maximum coverage by any test set T_j on S_i.

A possible metric for differences between the coverage on different SUTs is given by

$$c_i = \prod_{j=1}^{n} (cov_{ij})^{\frac{w}{n}} \tag{4.2}$$

with n being the number of SUTs and w being a constant which controls the impact of poor coverages on the *combined coverage* of a SUT. That means that SUTs which have a good coverage when executed with the test cases of the other competitors are benefitted and those which only are covered by a small fraction are punished. Only using a correction w without $\frac{1}{n}$ would not worsen the good coverage of only one test set.

The coverage is used as a weight for the *weighted performance* of a SUT:

$$p_i = \left(\sum_{j=1}^{n} \frac{p_{ij}}{(cov_{ij})^w} \right) / n \tag{4.3}$$

Again, n is the number of SUTs and w a control constant. For p_i, those test sets influence the performance value more which have a lower coverage. The impact of those weights can be controlled by w and the comparability is preserved by $\frac{1}{n}$. Especially, the weighted performance

benefits those SUTs which have a high coverage for every test set rather than those which only have a very high coverage for a few test sets. This effect is intensified by a higher value of w. The *combined coverage* and *weighted performance* are evaluated in Chapter 6.

4.1.1 Example

This section provides a simplified example for the usage of *combined coverage* and *weighted performance*. Figure 4.1 shows four different implementations of the very same Java method `max(int a, int b)` which should return the maximum of the two given integers `a` and `b`. The IBPC technique is used to determine which of the implementations performs best in comparison to the others. Note that the IBPC assumes that all given implementations are correct and fulfill the given requirements. Functional testing is not part of the IBPC.

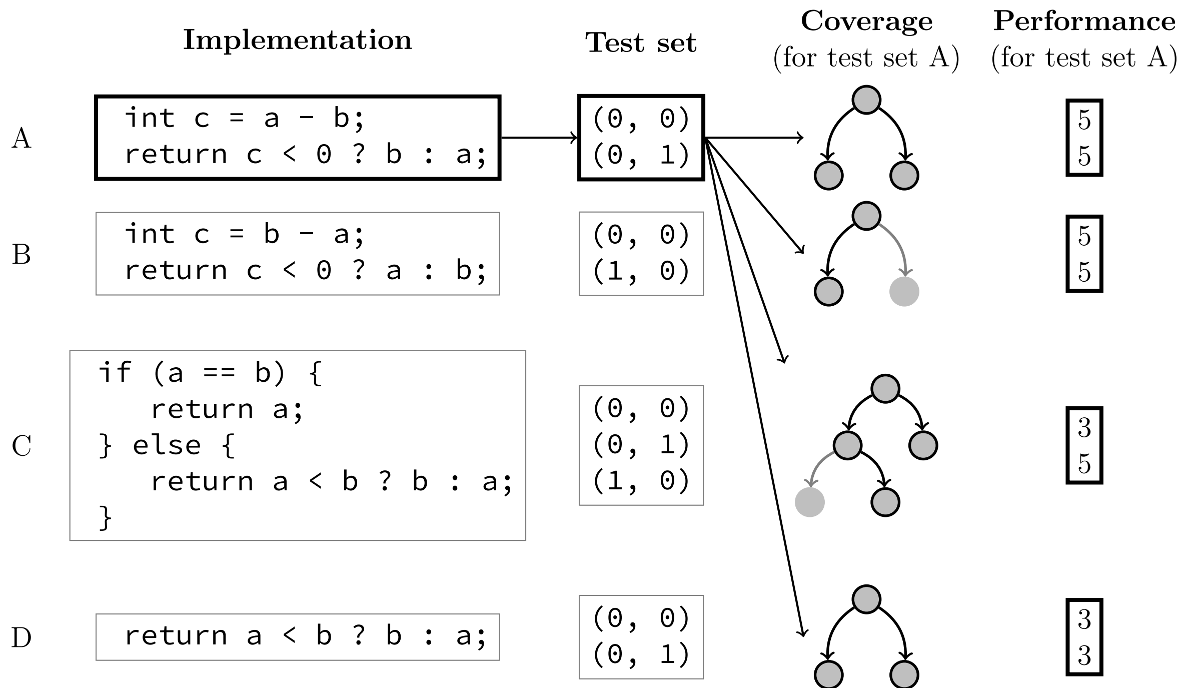


Figure 4.1: Example of the IBPC for four versions of a `max(int a, int b)` method. Test sets with maximized coverage are generated from each implementation (first to second column). Each implementation is executed with each test set and the coverage (third column) and performance (fourth column) are measured. The coverage is displayed by highlighting the covered nodes in the corresponding control flow graphs (third column).

In the first step, a test set with maximized coverage for each of the given implementations is generated. Here, a simple basic block coverage is used, with a basic block consisting of all byte code statements until the next jump statement. Each test set has a coverage of 1.0 on the respective implementation and is minimal in the number of required method invocations. For instance, in order to cover A, two invocations are needed. One where $a = b$ and the other with $a < b$. Implementation B also needs two invocations, one with $a = b$ but the other with $b < a$, to be fully covered.

Table 4.1: Basic block coverage per test set on each implementation (cov_{ij}).

executed on	test set generated for A	B	C	D
A	1.0	2/3	1.0	1.0
B	2/3	1.0	1.0	2/3
C	0.8	0.8	1.0	0.8
D	1.0	2/3	1.0	1.0

In the second step, the structural coverage and the performance of each test set executed on each implementation are computed. Here, basic blocks are used to represent the nodes of the CFGs. For a better visibility, Figure 4.1 only shows the coverage on each implementation by executing the test set generated for A. The coverage values for each execution are shown in Table 4.1. They can be interpreted as follows: Implementations A and D have the same control flow graph and therefore react equal on test cases. Implementation C has an additional test case caused by the first condition, the test on the equality of parameters `a` and `b`. Therefore, the test cases of C cover all other implementations, but C is not fully covered by the others. In contrast to A, C and D, implementation B returns parameter `b` if it equals `a`. That means that B can not be compared to the other implementations as good as A, C and D.

Table 4.2: Performance for each test case (left) and each test set (right) in number of executed byte code load and store operations (p_{ij}).

executed on	parameters for test case `(0,0)`	`(0,1)`	`(1,0)`
A	5	5	5
B	3	5	5
C	5	5	5
D	3	3	3

executed on	test set generated for A	B	C	D
A	5	5	5	5
B	5	5	5	5
C	4	4	$4.\bar{3}$	4
D	3	3	3	3

In addition, the performance of each test set executed on each implementation is computed. Here, instead of measuring the CPU time, the number of executed byte code load and store operations is measured as a deterministic and reproducible approach for this example. In order to exclude the possible influence of just in time (JIT) compilation in this example, it is disabled for the performance measurement. The impact of JIT compilation is studied further in Section 5.4. The number of byte code load and store operations for each of the used parametrizations performed on each of the implementations is shown in Table 4.2 together with the performance of each test set when executed on each of the implementations. Consider for instance implementation C. The comparison of the variables `a` and `b` requires two load operations, one for each parameter. Returning `a` requires another load operation, resulting in three operations for the case that `a` equals `b`. If they are not equal, the test for `a < b` requires two additional operations and returning `a` or `b` another load operation. Therefore a total of five operations is needed.

In the third step, the *combined coverage* and the *weighted performance* are computed. The results are shown in Table 4.3. The parameter w is set to two as an example and its impact

Table 4.3: Combined coverage and weighted performance with $w = 2$

implementation	combined coverage	weighted performance
A	0.816497	6.562500
B	0.666667	8.125000
C	0.715542	5.770833
D	0.816497	3.937500

is further discussed at the end of this example. As implementation B cannot be compared to the other implementations as good as A, C and D, it has the least combined coverage:

$$\begin{aligned} c_B &= \prod_{j \in \{A,B,C,D\}} (cov_{Bj})^{\frac{2}{4}} \\ &= (2/3 * 1 * 1 * 2/3)^{\frac{1}{2}} \\ &= 0.\bar{4}^{0.5} \\ &= 0.\bar{6} \end{aligned} \tag{4.4}$$

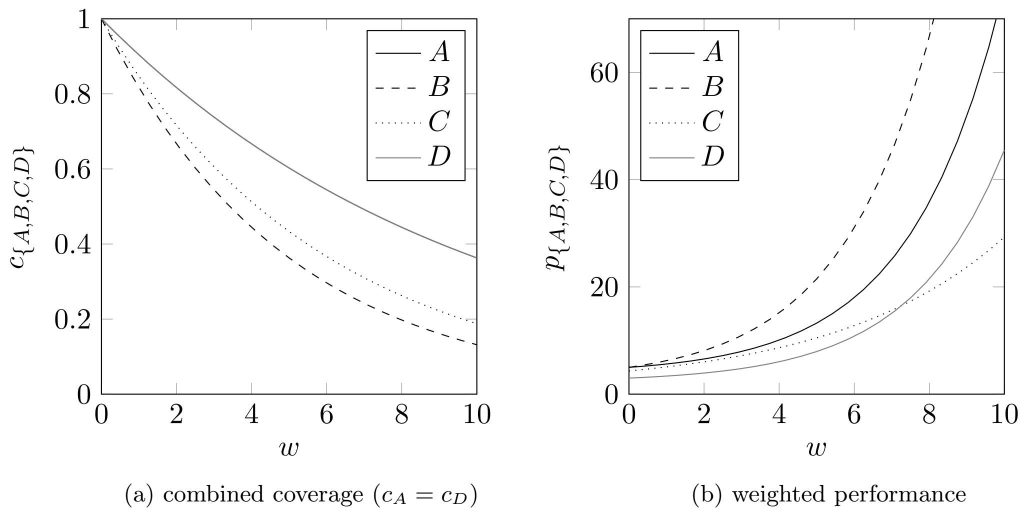
For the weighted performance one can observe, that D has the least and therefore best performance:

$$\begin{aligned} p_D &= \left(\sum_{j \in \{A,B,C,D\}} \frac{p_{Dj}}{(cov_{Dj})^w} \right) / n \\ &= (\frac{3}{1^2} + \frac{3}{0.\bar{6}^2} + \frac{3}{1^2} + \frac{3}{1^2})/4 \\ &= (3 + 6.75 + 3 + 3)/4 \\ &= 15.75/4 \\ &= 3.9375 \end{aligned} \tag{4.5}$$

This results from the fact that D has the least, three in total, number of byte code executions for any parameter. A and B use a local parameter `c` which adds two byte code operations for each invocation. The condition for equality in C makes every invocation of the `max(int,int)` method with equal valued parameters much faster in comparison to A and B. A and D have the best combined coverage but D has the better performance and is preferred.

Note that the parameter w is used to weight the influence of different coverage results. In fact, it should worsen the influence of a low structural coverage measurement more than it enhances the impact of one good result. Figure 4.2 shows the influence of different values for w on the combined coverage and weighted performance on the given example.

With increasing w, the combined coverage obviously decreases without changing the order of the different implementations. In case of the weighted performance, an increasing w not only increases the weighted performance but, resulting from the coverage weight, implementation D has a worse weighted performance than implementation C for $w > 7.5$. Keep in mind that a lower performance is considered to be a better performance. Here, implementation C performs best especially with an increasing w because it has the best of the worst coverage results ($0.8 > 2/3$).



(a) combined coverage ($c_A = c_D$) (b) weighted performance

Figure 4.2: Combined coverage and weighted performance for the given example of four implementations of the `max(a,b)` function.

4.2 Summary

The absence of a general accepted procedure to produce unbiased benchmarks and compare the performance of several competitors not with respect to a single feature but the complete functionality of the competitors make a new technique inevitable. For such an automated performance comparison, the Interface Base Performance Comparison (IBPC) technique is proposed. To the knowledge of the author, it is the first technique that does not concentrate on a single, user-defined set of features to compare the performance of a set of competitors (Section 2.3). In order to avoid benchmarks which do not cover all features of the SUTs and in order to penalize competitors which only perform well in a certain benchmark, the IBPC uses the *combined coverage* and *weighted performance*. Both metrics are new and exemplified by a simple example in this chapter. Their capabilities are evaluated in Chapter 6, where the outcome of an automated performance test comparison performed with an implementation of the IBPC, the Performance Test Automation Framework (PTAF, Chapter 5) is compared to the detailed performance analysis of the high-dimensional spatio-temporal indices in Chapter 6.

Chapter 5

The Performance Test Automation Framework

For the automated and unbiased comparison of a set of competitors, Chapter 4 introduces the new Interface Based Performance Comparison (IBPC) technique. Here, a competitor is the implementation for the solution of a specific task, e.g. the efficient indexing of spatial data. An implementation of the IBPC technique, the Performance Test Automation Framework (PTAF) is presented in this chapter. First, a general overview of PTAF is given, then the concept and concrete implementation of the modules for code coverage, test generation and performance measurement are discussed in detail. In order to get a good overview over the required tasks, modules and their communication with each other, Figure 5.1 highlights the main stream of the control and data flow in PTAF with an UML activity diagram.

It shows that only the byte code of the competitors and a common interface is needed to compute the combined coverage and weighted performance. Both metrics may be used to draw conclusions about the performance of a system in relation to other competitors. The general workflow of the implemented framework is as follows. Given the byte code of all competing SUTs and a common interface, the byte code is instrumented and the corresponding CFG is extracted (*instrument*). The instrumented byte code and the CFG are not only used for the measurement of coverage (*measure coverage*), but also for the generation of test sets with a high coverage (*generate best covering test sets*). The generated test sets are executed on the unmodified byte code and the performance of the competitors is measured (*measure performance*). The coverage and performance measurements are then used to compute the combined coverage and the weighted performance. The actions *compute combined coverage* and *compute weighted performance* are defined in Section 4.1. The actions *instrument*, *measure coverage* (Section 5.1), *generate best covering test sets* (Section 5.2) and *measure performance* (Section 5.4) are described in detail in the following sections. Note that all of these modules are exchangeable but language dependent. As a reminder, PTAF is programmed in the Java 8 programming language.

5.1 Instrumentation and Code Coverage Measurement

This section describes the instrumentation of the given byte code, the extraction of the corresponding CFG and the computation of the coverage. The existing code coverage tools for Java are discussed in Section 2.9. PTAF is a prototype implementation such that the actual

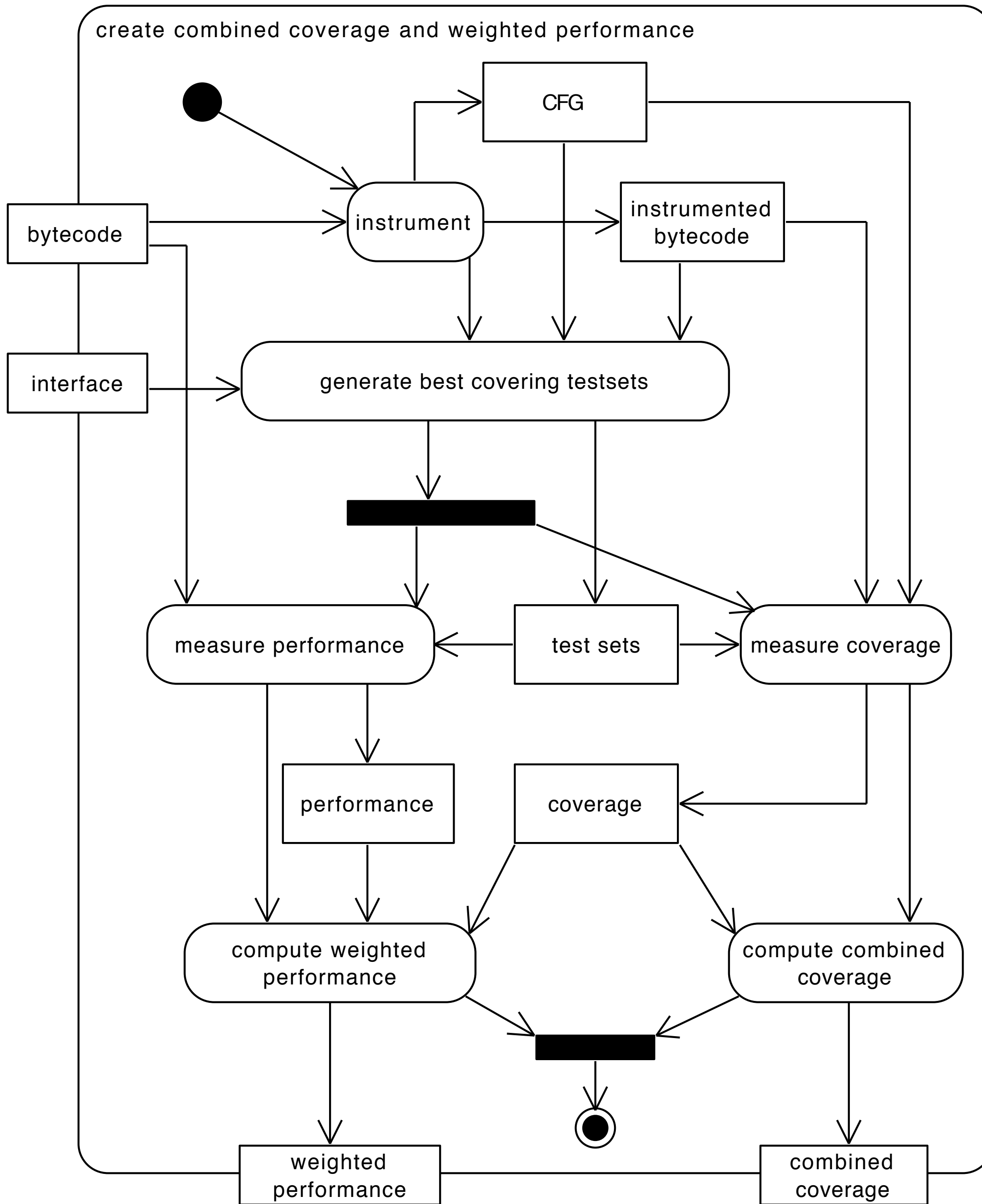


Figure 5.1: UML activity diagram of PTAF.

performance of the computation itself is of lesser concern. Also, following the prototypal approach, the instrumentation and coverage procedure needs to be flexible and adaptive to new coverage types. The existing code coverage tools are bound to the coverage types listed in Table 2.3 and are not designed to be extended by additional coverage types. In addition to the required flexibility, not only the abstract coverage results are needed but the corresponding control flow graph for a more detailed analysis. These requirements make a new coverage tool inevitable.

5.1.1 Instrumentation and CFG Detection

The concepts of code instrumentation in Java are described in Section 2.10. Although the overall computation time of the coverage tool is of lesser concern, choosing a fast working approach does not need to be a drawback. Therefore, byte code instrumentation is preferred over source code instrumentation. In order to minimize compilation and loading errors in the Java Virtual Machine, on-the-fly byte code instrumentation using a Java agent is chosen. Doing so, the instrumentation procedure will not miss to instrument any class.

The basic idea of the new instrumentation approach is that the inserted probes are directly linked to the control flow graph. As a reminder, a probe is a set of byte code statements that are inserted in the byte code and used to track the execution of certain parts of the byte code. Here, each probe represents a certain control flow node. If the virtual machine accesses a certain probe, it invokes the linked coverage generator module. Depending on what type of coverage generator has been chosen, the invocation through the probe is handled differently. A statement coverage generator may only record one access. A branch coverage generator at least needs to store the last accessed node in order to record which branch has been traversed. Those invocations are very expensive. For instance, EclEmma [101] instruments each class with a byte array, with each field representing the number of accesses on one node which is much more efficient. For each access only one array field has to be incremented instead of an invocation followed by an unknown number of additional operations caused by the coverage generator. In contrast to the new approach described here, EclEmma may not be extended to record path or decision coverage. In addition, thread safety or the flexible incorporation of several different coverage generators may not be applied on EclEmma.

The instrumentation process works in two phases: a record phase and an instrumentation phase. A complete example is given in Figure 5.2 to show the instrumentation and the corresponding control flow graph (CFG) of a simple `max(int,int)` method. With the start of the instrumentation process, the record phase begins. During instrumentation and probe insertion, the corresponding CFG is built up and the byte code of a single method is parsed sequentially. The targets of some jump instructions may be before the actual jump instruction, i.e. the target of a control flow edge must be known prior to the source of an edge. For instance, in the given example, the first conditional jump in line 10 `IF_ICMPGE L1` requires target label `L1` before it is actually parsed. Therefore, all targets which are the targets of a jump instruction are collected throughout the record phase. The record phase will also collect the begin of all catch blocks, as they are considered to be entry points in a method. After the invocation of another method, if the invoked method throws an exception which is caught, the catch block in the invoking method needs to be differentiated from the rest of the methods CFG. In addition, the end of the method will be stored in order to determine the end of the actual instrumentation process.

After the end of the record phase, the instrumentation phase begins. A probe always consists of the invocation of an abstract method which may be implemented by one of the coverage generators. Each probe invokes the same method. The method is invoked with a unique ID representing the corresponding CFG node. The ID is not only unique for one method or class but for all classes that are instrumented throughout the complete instrumentation process. This is required because an execution trace may jump between several classes when method invocations or try-catch blocks are executed by the virtual machine. As a drawback, the results of several independent instrumentation runs may not be combined with each other. Nonetheless, given a deterministic system under test (SUT), the instrumentation process works

```
1  max(II)I
2     LDC 3
3     INVOKESTATIC CoverageConnector.visitMethod (I)V
4    L0
5     LINENUMBER 34 L0
6     LDC 3
7     INVOKESTATIC CoverageConnector.visit (I)V  //  if(a < b)
8     ILOAD 1
9     ILOAD 2
10    IF_ICMPGE L1                                    true
11   L2
12    LINENUMBER 35 L2
13    LDC 4
14    INVOKESTATIC CoverageConnector.visit (I)V  // return b;     false
15    ILOAD 2
16    IRETURN
17   L1
18    LINENUMBER 37 L1
19   FRAME SAME
20    LDC 2
21    INVOKESTATIC CoverageConnector.visit (I)V  // return a;
22    ILOAD 1
23    IRETURN
24   L3
25    LOCALVARIABLE this LMax; L0 L3 0
26    LOCALVARIABLE a I L0 L3 1
27    LOCALVARIABLE b I L0 L3 2
28    MAXSTACK = 2
29    MAXLOCALS = 3
```

Figure 5.2: Example instrumentation of a simple `max(int a, int b)` Java-method. Before the first byte code instruction of each basic block, an invocation to the static method `visit(int)` with the ID of the corresponding CFG node is inserted. A certain coverage generator may register to be notified for any invocation of `visit(int)`. The probes are presented **bold**. The **LDC** instruction is required by the Java virtual machine.

deterministic, i.e. the instrumentation of the same SUT will result in the very same IDs for the CFG. Note that the numbering of the nodes in Figure 5.2 starts with 2, as 0 and 1 have already been spent for the identification of the nodes in the default static initialization method of the instrumented class. The node ID 2 is invoked after 4 because the corresponding node has been identified as the target of a jump instruction in the record phase.

The probe, i.e. method invocation, is inserted before the first instruction of a certain byte code block with a block being a series of byte code instructions without a jump. A new CFG node is created and the corresponding probe is inserted if...

- ...the first label of the CFG is parsed.
- ...a previously recorded target is parsed.

- ...the last node that has been parsed is a jump, lookup-switch or table-switch instruction.

- ...the parsed instruction is a throw or return instruction. If the block contains a method invocation instruction, it is split up into two blocks in order to separate the throw/return from the invocation. The separation of throw and return instructions is important because one needs to be sure if the method has been completely executed. This is especially useful to determine the number of executions of a recursive method. Recursive methods are needed for the generation of suitable performance tests in Section 5.2.2.5.

Besides the construction of the CFG nodes, the edges between the CFG nodes need to be created. An edge has no equivalent on the byte code level but only in the corresponding CFG. An edge is created between...

- ...two successive nodes if the last instruction of the previously created node is not a goto instruction. A goto is an unconditional jump instruction.

- ...a node whose last instruction is a jump, lookup-switch or table-switch instruction and all nodes which are targeted by the jump instruction. The targets are recorded in the record step.

The first node and all nodes which start with a catch instruction do not have any incoming edges. All nodes whose last instruction is a throw or return instruction have no outgoing edges. Edges are unambiguously identified by the unique IDs of their source and target nodes.

In order to record the control flow between instrumented methods and recursive method invocations, at the start of every method a certain `visitMethod(int)` method is invoked. If a coverage generator registers for updates on this method, it may, for instance, record recursive method invocations.

Summing up, the instrumentation process merges all continuous byte code instructions without a jump instruction to a basic block. At the beginning of each basic block, a probe is inserted which invokes a certain method with a unique node ID. The ID represents a certain node in the CFG that is built up. The nodes in the CFG are then connected by edges. The jump instructions in the byte code denote which nodes need to be connected.

5.1.2 Coverage computation

The implementation of the instrumentation system and the generation of the CFGs is described in the previous section. This section describes how the instrumented byte code and the CFG may be used in order to compute several coverage types when the instrumented code is executed. An overview of the class structure is given in Figure 5.3 and a simplified sequence of the execution of the instrumented `max(int,int)` method is depicted in Figure 5.4. In order to calculate the coverage, the desired coverage generator has to be connected to the `CoverageConnector` whose `visit(int)` method is invoked by the instrumented probes. Once an instrumented method is executed, for each basic block that is reached this `visit(int)` method is invoked and it invokes the `visit(int)` method of the registered coverage generator. After the execution is completed, one may get the coverage by invoking the `calculateCoverage()` method in the registered coverage generator.

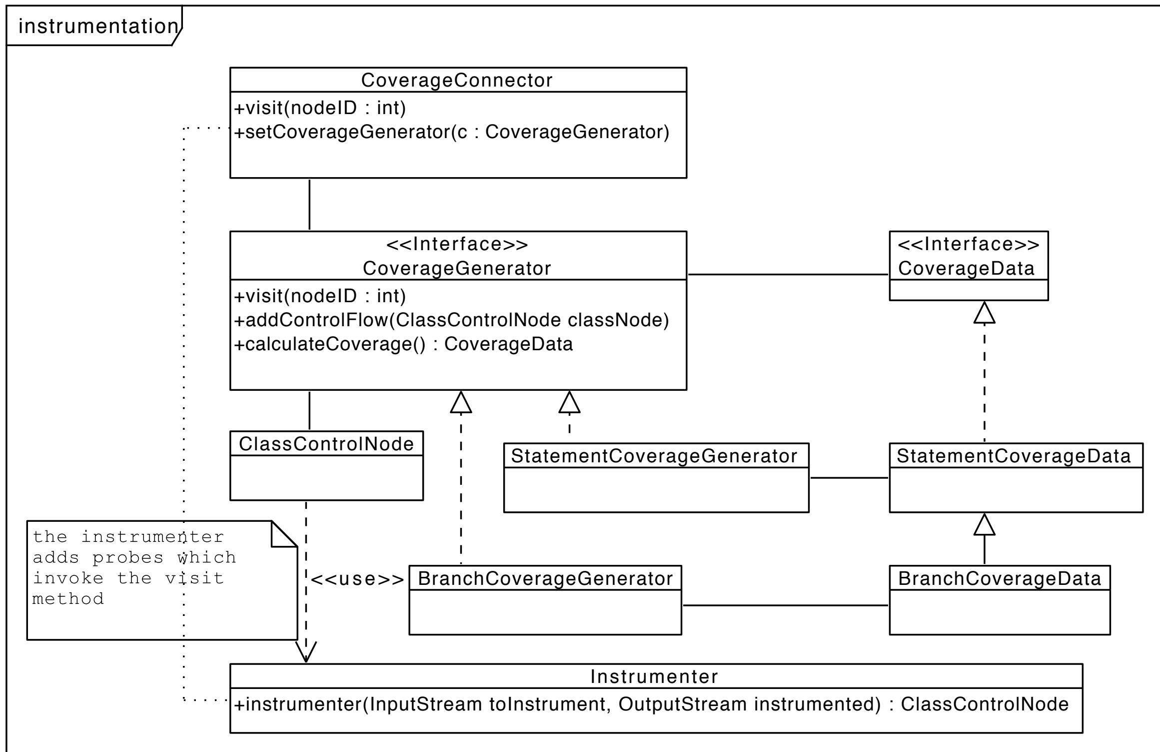


Figure 5.3: UML class diagram for the basic structure of the instrumentation and coverage system.

5.1.2.1 Statement (Basic Block) Coverage

The different understandings of statement coverage in the literature are discussed in Section 2.9. Here, one node represents all statements that are always executed without being interrupted by a jump. The statement coverage generator simply records the execution of a node with every access. The fraction of the nodes accessed at least once to the total number of nodes in the CFG denotes the *basic block coverage*. The number of statements a single node is representing may easily be recorded in the instrumentation phase. One determines the *statement coverage* by summing the number of nodes for every statement which is at least accessed once and dividing it by the total number of statements in the computed CFG.

5.1.2.2 Branch Coverage

Computing the branch coverage simply means to determine the number of branches which are accessed at least once and dividing it by the total number of branches in that CFG. If a CFG consists of only a single node and no branches, it is considered to have full branch coverage if the node is accessed and zero branch coverage if that node is not accessed. A branch is considered to be accessed if its source node and target node have been accessed one directly after another. Naturally, with the computation of the branch coverage, also statement and basic block coverage may be computed.

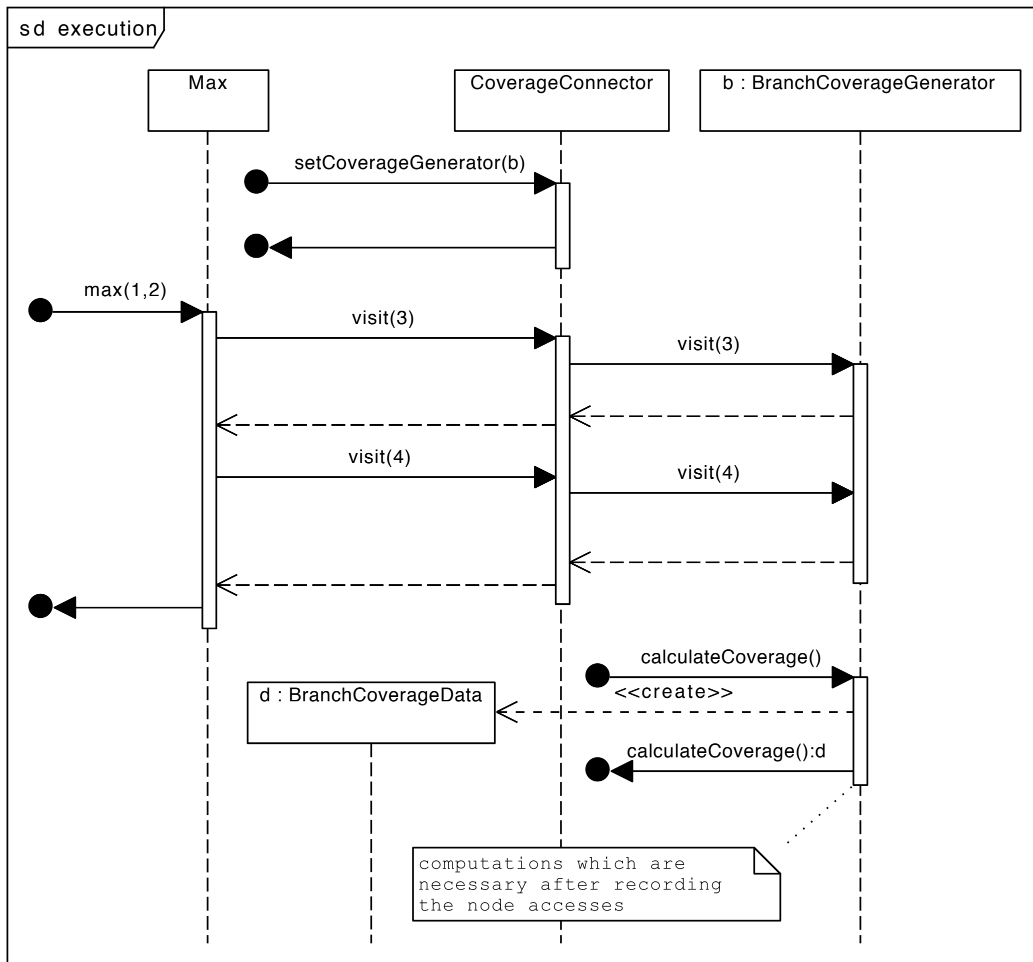


Figure 5.4: UML sequence diagram describing the basic workflow of the coverage framework. `CoverageConnector` is accessed in a static way.

5.1.2.3 Loop Coverage

Loops in a CFG are identified by a header and a back edge. Since every node may be a header, the back edge may be identified by using the dominator relationship (see Section 2.9). In the new Java coverage tool described here, the loops of a CFG are identified by Algorithm 4. It traverses all nodes in a CFG g and searches for those nodes which are dominated by node n. For each node m which is dominated by n it looks for a back edge $(m, n) \in g$. If such a back edge exists, a loop is found. The algorithm has an asymptotic runtime behavior of $O(n^2)$ with n being the number of nodes in the CFG and may be a part of optimization when a more efficient coverage computation is required. As the header of a loop is the target of the back edge, a loop may solely be identified by its back edge.

The loop coverage is recorded by two values. First, the number of accesses n_a of a loop describes the number of accesses of the loop header n which is already recorded by a statement or branch coverage generator. The number of loop iterations n_i is recorded by the number of accesses of the loops back edge. This is already done by a branch coverage generator. The loop

```
Input  : CFG g, Map doms(m, d) of nodes m and corresponding dominators d
Output: all loops in g
1  foreach node n ∈ g do
2      foreach d ∈ doms do
3          if n ∈ d then
4              if g contains edge (m, n) then
5                  loop with header n and back edge (m, n) found;
6                  record loop;
7              end
8          end
9      end
10 end
11 return all loops;
```

Algorithm 4: Simple algorithm to find all loops in a CFG using the dominators of each node.

coverage may also be determined in relation to the maximum of loop iterations. Consider a back edge $(m, n) \in g$ and a user defined value of three loop iterations that are desired until that loop is considered to be fully covered. The relative loop coverage for that loop would than be $\frac{i}{3}$ with i being the number of loop iterations. This relative loop coverage is used by the fitness function described in Section 5.2.2.5 and will be discussed in more detail there. Summing up, aside the identification of the CFGs loops, no additional computation in comparison to a branch coverage generator is required for the computation of the loop coverage of a CFG.

5.1.2.4 Path Coverage

Meyering (2016) [136] implements a path coverage generator on top of the new coverage generator presented here. It consists of an algebra to describe a path more compressed. For instance, four iterations of a loop a are described by a^4. This algebra also may be used to calculate the maximum number of possible paths in a CFG. The maximum number of iterations of one loop may be limited to a certain value. Although this approach limits the problem of path explosion, the paths have to be recorded anyway. For the record of all executed paths, a trie, i.e. prefix tree, is used. This reduces the required space but nonetheless the recording of paths exceeds a reasonable amount of space. For instance, the recording of the path coverage of a single workload as described in Section 3.5.1 takes over 1 TB disk space when stored in a space saving binary format.

5.1.2.5 Condition Coverage

Vatterodt (2016) [196] implements the basics for a condition coverage generator by extending the instrumentation and CFG detection. During the record and instrumentation phase, the conditions of each decision are recorded and for each decision a sub graph of conditions is built. Using this sub graph, the evaluated conditions can be identified and it can be recorded which conditions are covered by an execution. The condition coverage generator is able to compute simple condition coverage, minimal multiple condition coverage and multiple condition coverage. The initial set up of the CFG and the instrumentation for the condition coverage

generator is much more complex than for the basic statement, branch and loop coverage generator described above. Nonetheless, during the execution of the byte code, no additional computation costs are expected for the condition coverage generator in comparison to the basic coverage generators.

5.2 Automated Test Generation

This section describes the implementation of an automatic generator of test sets which are derived from a predefined set of methods and initializers. Section 5.2.1 evaluates the ability of the existing test generators to generate such test sets and concludes that a new generator is necessary. This new approach is described in Section 5.2.2.

5.2.1 Evaluating Existing Test Generators

In this section the existing test generators outlined in Section 2.11 are used to create test sets with a high structural coverage only by using methods and initializers from a given interface. Several implementations of the R-tree and its variants and their conversion to a comparable format are described. In addition, the test generators and especially the alterations of these test generators which are needed to let them generate test sets on basis of a single interface and an SUT are described. This section indicates whether a new test generator for interface based test sets is needed or an existing test generator can be chosen.

The R-tree is chosen because of its widespread usage and the similarity of indices for spatial data to indices for spatio-temporal high-dimensional data. Indices for high-dimensional spatio-temporal data are not chosen exclusively for the evaluation of existing test generators as only implementations by the author of this thesis exist. A test generator may not be suitable for a specific implementation style used by a single person. In order to avoid such a bias, the SUTs used for the evaluation should be diverse in their implementation style but have a single interface in common. Using R-tree variants is therefore a good compromise between a diverse implementation style and comparability to the actual investigated high-dimensional spatio-temporal indices.

Table 5.1: Different implementations of the R-Tree and its variants used for comparison and evaluation throughout this thesis.

Name	Source	R-tree type				License
		linear	quadratic	R*	reinsert	
XXL for Java	[24]	✗	✓	✓	✗	GPL
Android-R-Tree	[195]	✗	✓	✗	✗	Custom
Java Spatial Index (JSI)	[10]	✓	✗	✗	✗	GPL 2.1
David Moten's R-Tree	[139]	✗	✓	✓	✗	Apache 2.0
Russ Week's R-Tree	[202]	✓	✓	✗	✗	LGPL 3.0
Menninghaus' R-Tree	this thesis	✓	✓	✓	✓	

Table 5.1 lists the libraries used for the evaluation here and their supported R-tree variants. In addition to the R*-tree's split algorithm, only the implementation of the R*-tree by the author of this thesis (Menninghaus R-tree) uses the forced reinsert (Section 2.2.1) as an

additional overflow strategy. For a better comparability, the implementation by the author of this thesis offers two R*-tree implementations: with and without the forced reinsert. The XXL, Android, David Moten's and Russ Week's library do only support an implementation of the intersects, i.e. *overlaps* query. Therefore, the *contained* query is implemented anew by the author of this thesis for these indices on basis of the given query implementations. The *contained* query is selected as the query type which has to be supported by all indices for a later comparison with the results from Chapter 3 which are also based on the *contained* query. As the Android, JSI and David Moten's library only support 2-dimensional rectangles, the complete evaluation is limited to 2-dimensional rectangles. In contrast to Chapter 3, every implementation is set up without a previous optimization step but with a fixed maximum node size of four and a minimum node size of two elements. As a reminder, a node with less than the minimum node size of elements has to be merged and a node with more than maximum node size elements has to be split (Section 2.2.1). Doing so, the probability for split and merge is maximized and fewer elements need to be inserted in order to reach different states in an index. Each implementation is solely chosen because of its availability. The set of the given implementations may not be complete but is sufficient for the purpose of the evaluation of test generators and the evaluation of PTAF.

Each of the implementations is integrated into the evaluation system by using a wrapper class which implements the `NDRectangleKeyIndex`-interface (Figure 3.6). Doing so, the test generators can generate test sequences based on the same interface for each of the implementations. This is one of the core ideas of the IBPC. In general, the set of methods and initializers, which may be used by the test generators to generate test sequences, is listed in the following:

- the default-initializer for a certain `NDRectangleKeyIndex`
- `<init> NDPoint(double[])` initializes a new d-dimensional point by the given `double`-array
- `<init> NDRectangle(NDPoint, NDPoint)` initializes a new d-dimensional rectangle by the given d-dimensional lower left and upper right corners
- `<init> Java .lang.Object()` initializes a new Java-Object
- `ObjectReference.getReference(Object)` returns a unique reference for the given Java-Object
- `<init> NDRectangleKey(NDRectangle, ObjectReference)` initializes a new key-value pair, with a d-dimensional rectangle as key
- `<init> DefaultRectangleQuery()` initializes a new default query
- `NDRectangleKeyIndex.insert(NDRectangleKey)` inserts a new key-value pair into the index
- `NDRectangleKeyIndex.delete(NDRectangleKey)` deletes an existing key-value pair from the index
- `NDRectangleKeyIndex.getContained(NDRectangle,RectangleQuery)` invokes the given query for any key in the index which is contained in the given rectangle
- initialize a `double` value or an array of `double` values

This set of methods and initializers is derived from the implementation of the STPA and RST-tree, described in Section 3.4. It is the minimum set of methods and initializers which is required to build up and fill an index structure and to update, delete and query its elements. The update and deletion of elements in an index is necessary to create node structures which may not be created by insertion only. Following from Section 2.2, not only the split but also the merge of several nodes is crucial to generate different structures. Note that the first entry in this set, the default-initializer, is varying for the implementation for which the test sets should be generated. For instance, a valid test sequence for a concrete implementation of a 2-dimensional `RTree` could look like in Listing 1.

```
1  RTree rtree = new RTree();
2  double d1 = 0.0;
3  double d2 = 1.0;
4  NDPoint point1 = new NDPoint(d1,d1);
5  NDPoint point2 = new NDPoint(d2,d2);
6  NDRectangle rectangle = new NDRectangle(point1,point2);
7  Object object = new Object();
8  ObjectReference reference = ObjectReference.getReference(object);
9  NDRectangleKey key = new NDRectangleKey(rectangle,reference);
10 rtree.insert(key, reference);
11 DefaultRectangleQuery query = new DefaultRectangleQuery();
12 rtree.getContained(rectangle,query);
13 rtree.delete(key);
```

Listing 1: Example of a method sequence for the generation and test of a simple concrete implementation of a 2-dimensional R-Tree.

For each of the implementations, a set of classes, the SUT, is defined. The general goal of each test generator is to maximize the coverage of the generated test sets on this SUT. The elements of the SUT are selected before the start of the test generation by analyzing the source code and identifying all classes within the given library which are affected by the invocation of the elements of the interface. These classes are then added to the corresponding SUT.

Following from the test sequence example and the given interface above, the evaluated test generators should be able to

- identify sub- and super-types in order to generate the index by the given default-initializer but access it through the methods of the interface.
- identify valid points and rectangles, i.e. only generate 2-dimensional points and rectangles.
- maximize the coverage not only of one single class under test but a complete SUT.
- only use the elements of the given interface and no other elements for the generation of the test sets.
- produce non-failing test sets, i.e. test sets that are compilable and do not throw exceptions.

Tables 2.6 and 2.7 list the existing test generators discussed in Section 2.11. Only test generators which are publicly available at the end of 2017 and fit the desired requirements

are taken into consideration: RANDOOP, TestFul, EVOSUITE, T3, MOSA, JTExpert, GRT and DynMOSA. T3, MOSA, JTExpert and GRT fail with an Exception during the generation of any test set. TestFul produces the desired test sets but they are not compilable as every instance is referenced as a plain Java `Object` but invoked with the methods from the SUT without the required type cast. Only four generators produced the desired results: non-failing test sets solely on basis of the given interface while maximizing the coverage on the complete SUT: RANDOOP, EVOSUITE, MOSA and DynMOSA.

As the implementations of MOSA and DyMOSA are branches from the EVOSUITE implementation, all three are adapted in the same manner. In order to aim for a maximized coverage on the complete SUT and not only a single class, the coverage goal is extended to all classes in the SUT. The extension of the coverage goal requires a manipulation of the EVOSUITE implementation which is taken from Graf (2017) [91] for the branch coverage goal and adapted for the other coverage goals by the author of this thesis. The automatic detection of methods and initializers in EVOSUITE which is used to generate test sets with a high structural coverage, needs to be limited to the given test interface. This limitation is achieved by designing wrapper classes for each index which only contain the desired methods and do not inherit other classes, i.e the wrapper classes only inherit from the plain Java `Object`. The execution is limited by the total computation time which is set to 5 and to 30 minutes. The evaluation is performed on a Dell PowerEdge R420 with 192GB 1,600Mhz DDR3L RAM, Intel Xeon E5-2420 CPU and four SATA 7,200rpm hard disk drives. For one evaluation set, all other parameters are set to the default configuration as recommended in [82]. A lot of time and effort has been spent in this thesis to optimize the various parameters offered by EVOSUITE and its successors MOSA and DynMOSA. Besides the default configuration of EVOSUITE, one other configuration is shown here to give an understanding about the impact of different configurations. Note that EVOSUITE offers 339 different parameters which can not be explained in more detail here. The default configuration of EVOSUITE is changed as follows:

- The maximum length of a chromosome, i.e. method sequence, is enlarged from 10 to 1000 in order to support more complex test sequences.
- The maximum depth of inheritance is enlarged from 3 to 10 such that the test generator also accepts types which are inherited over more than three steps.
- The local search budget, i.e. the time being spent to optimize the parameters of a single method invocation, is enlarged from 5 to 40 ms and multiplied by the number of test goals. Doing so, the local search budget for a test sequence is enlarged depending on the complexity of the SUT.

In order to avoid a system crash due to limited resources, the output of RANDOOP is limited to one billion tests per run and only tests are generated which do not reveal errors. The default configuration of RANDOOP is not changed in any other way. For comparison, the coverage for a classic workload generator is computed. The classic workload generator from Section 3.5.1 is altered as follows: Every time the workload generator extends an existing history, the alteration is computed without resolving the bi-temporal inclusions. That is, with a deletion, an element is removed from the index, the update changes an existing element and so forth. As the computation of the coverage is very time consuming, the total number of inserted elements is reduced to 10000 elements from the original 100000. The results in

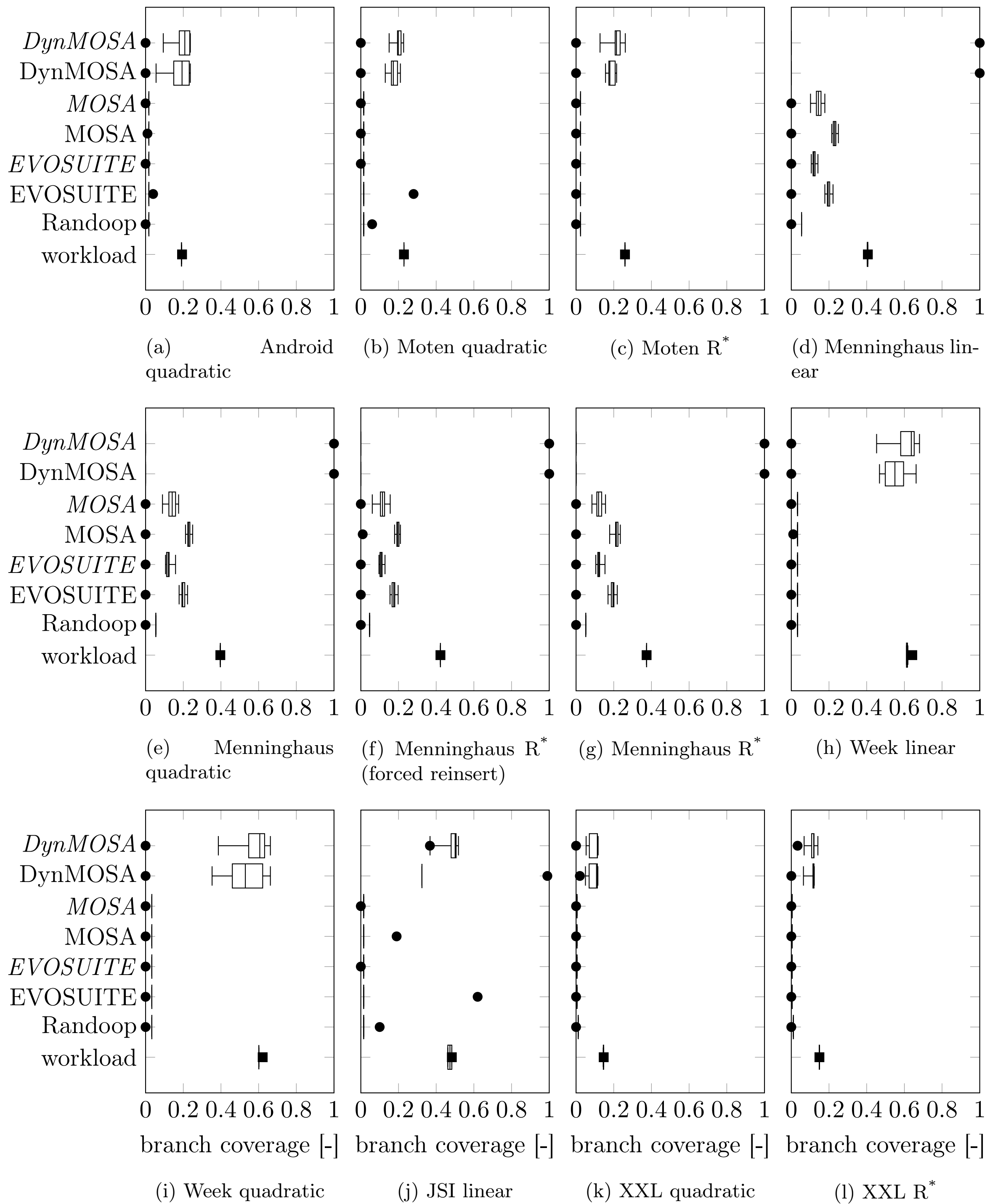
Section 5.2.2.5 indicate that the decreasing of inserted elements does not influence the branch coverage because there is little to no variance between the workloads. As an additional proof, the branch coverage of all workloads with a total of 10000 inserted elements and the branch coverage of all workloads with a total of 1000 elements is computed. Both workload groups do not only have the same branch coverage but cover the very same branches. As a matter of fact, the number of branch accesses and the number of loop iterations varies for a different number of inserted elements. The relation between the coverage and the number of elements inserted into an index will be discussed further in Section 5.2.2.5.

5.2.1.1 Results

This section shows the results for the test generation with all test generators. A total of approximately nine person months work has been spent to build up the evaluation system and especially test, integrate, adapt and configure the different test generators. All computations are performed on a Dell PowerEdge R420 with 192GB 1,600Mhz DDR3L RAM, Intel Xeon E5-2420 CPU and four SATA 7,200rpm hard disk drives and the pseudo random numbers are generated on basis of a random seed making the evaluation reproducible. The evaluation compares the branch coverage for all test generators and the classic workload generator (Section 3.5.1). For each implementation, a plot with a box plot for each test generator and the workload generator is displayed. The lower whisker shows the 2.5 percentile, the upper whisker shows the 97.5 percentile. The box is bounded by the 25 and 75 percentile, the line in the middle denotes the median. For each test generator, the dot denotes the fraction of generated test sets with zero coverage. All test generator executions with zero coverage are removed from the evaluation. The square at a workload generator denotes the coverage of all workloads generated by that workload generator run. Each workload generator and each test generator has been executed 100 times. The pseudo random numbers used for each workload and each test generator base on a predefined random seed. Thus, the evaluation becomes reproducible.

Figure 5.5 shows the branch coverage of the classic workload generator and the four different test generators. For instance, Figure 5.5 shows the branch coverage of the test sets created by the four test generators and the branch coverage of the classic workload generator for the R-tree from the Android-R-tree library ([195]) using a linear split algorithm. The dots on the left side of the plots show that nearly no test set has a branch coverage of zero. In general, the classic workload generator has the highest or nearly highest coverage results for each implementation. It is outperformed by DynMOSA for the Android and Russ Week's implementations but only for the maximal achieved coverage by DynMOSA. DynMOSA is unable to produce test sets which cover anything for all implementations from the author of this thesis. In contrast to DynMOSA, the other generators produce test sets with a coverage comparable to the coverage which is achieved for the other implementations. The coverage of all workloads together is only slightly higher than the coverage of the single workloads. In addition, the variance for the workload generator is very small. The custom configuration made on EVOSUITE, MOSA and DynMOSA only shows a slight variation in the results and is within the results of the default configuration. The same observations can be made for a runtime of 5 and 30 minutes. An even shorter runtime of 2 minutes causes a lot of additional failing tests as EVOSUITE and its successors need some time for set up and their warm up phase.

The results indicate that a new test generator which is specialized for the generation of interface based test sequences is necessary to implement the IBPC efficiently. Especially the



(a) Android quadratic (b) Moten quadratic (c) Moten R* (d) Menninghaus linear

(e) Menninghaus quadratic (f) Menninghaus R* (forced reinsert) (g) Menninghaus R* (h) Week linear

(i) Week quadratic (j) JSI linear (k) XXL quadratic (l) XXL R*

Figure 5.5: Branch coverage of twelve R-Tree implementations for a classic workload generator and four different test generators with default settings. Italic generator labels depict generators with the custom configuration. The dots denote the fraction of test sets with zero coverage. All generators have a time limit of 30 minutes, the results for a time limit of 5 minutes are identical. One workload contains 11000 insert, delete and update operations.

implementation of the IBPC may only be compared to the classic workload generation approach if the performance tests cover at least the same amount of code. This new generator may be partly derived from DynMOSA which shows the best results among the test generators. However, one must keep in mind that DynMOSA failed creating test cases for the implementations by the author of this thesis.

5.2.2 Automated Test Generation based on an Interface

This section uses the requirements summed up in the previous Section and the conclusions drawn from the existing approaches on automatic test generators in Section 2.11. Following from Section 2.11, it is obvious that the current "state of the art" of automatic test generation is to use genetic algorithms. The evaluation of the existing approaches used for the generation of interface based test generators uncovers the necessity of a new generator. Although genetic algorithms seem to be the best choice for automatic test generation, it can not be determined how the countless parameters that can be optimized for those algorithms may be configured. Even the setup of the chromosomes, the selection, recombination and mutation algorithms used by a genetic algorithm may be adapted to the specific problem. As the reviewed literature only offers insights on how the setup of a genetic algorithm for test generation is made and not how those algorithms react on changes in the configuration, the generator described here is built upon the requirements and later evaluated as a whole. A detailed evaluation on all settings and the influence of single modules would exceed the scope of this thesis. As long as the generator produces test sets which cover the SUTs at least as good as the classic workload generator, it fulfills its task. The new generator described in this section is not only designed to perform well on the generation on test sets for spatio-temporal high-dimensional indices but on complex data structures which provide an interface in general. Therefore, it is referred to as interface based test generator (IBTG) in the following.

For the implementation of the genetic algorithms, the multi objective evolutionary algorithm (MOEA) framework [96] is used. The MOEA framework already provides a good portion of the currently used genetic algorithms and operators which may easily be incorporated for the use with a custom problem representation as the one described in the following section. The basic input for the described generator is the test interface. The interface contains all methods and initializers that are needed to build up the desired sequences. An example of such an interface is given on page 92. It is up to the user to create a consistent interface, i.e. an interface where any type required by the given methods and initializers is either a primitive type, a type that is returned by one of the methods and initializers or an array of those types. Although the genetic algorithms and selection operators are provided by the MOEA framework, the representation of the test sets, the mutation and recombination operators and the fitness function need to be created to reflect the requirements of the IBPC: test sets with a high structural coverage generated on the basis of an interface. The following sections describe the new test set representation, the mutation and recombination operators and the fitness function which are then used by the genetic algorithms. For the general definition of genetic algorithms, see Section 2.11.4.1.

5.2.2.1 Representation of Test Sets

As a genetic algorithm is used, the first question is how to represent the desired test sets in the genetic algorithm. For genetic algorithms, the literature offers countless approaches for the

selection, recombination and mutation of binary representations. Nonetheless, a non-binary representation is chosen and the algorithms for selection, recombination and mutation are re-implemented. Such a specialized representation offers the possibility of using non-binary operators, i.e. operators whose function directly emerges from the intended structure of the test sets and the requirements. In addition, the non-binary representation may also be used to execute the test sets. Doing so, the test sets do not need to be converted from a binary format to the executable test set representation after each iteration of the genetic algorithm. Figure 5.6 depicts the structure of the representation of a test set in an UML class diagram. These classes are also used for storage and execution.

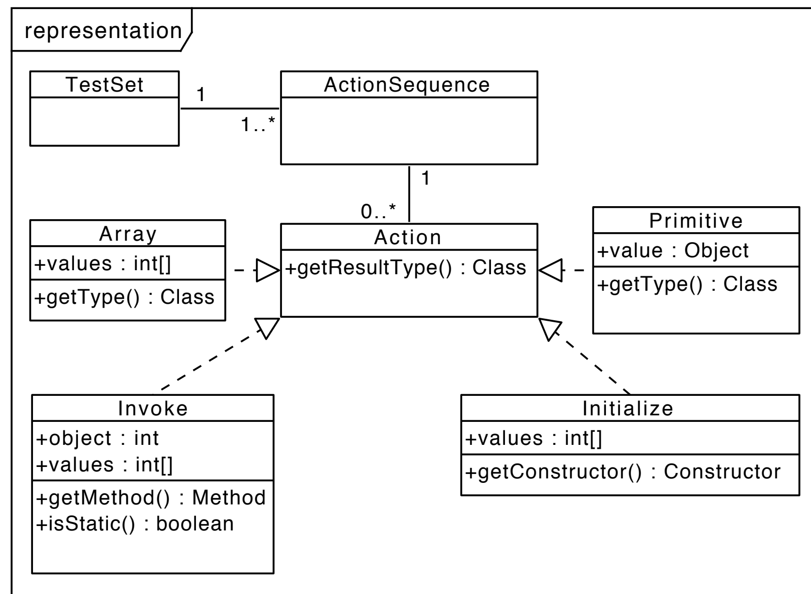


Figure 5.6: UML class diagram of the classes representing a test sequence in PTAF.

A `TestSet` consists of $n \geq 1$ `ActionSequence` instances. An `Action` is one atomic operation in a method sequence which is represented by an `ActionSequence`. All `Action` instances refer to objects created during an `ActionSequence` by the index of their creation operation in the sequence. For instance, Table 5.2 shows the representation of the example method sequence from Listing 1. The initialization of the R-tree is set at the start of a sequence at position 0, all other `Action` instances will refer to the R-tree as object 0, e.g. the `delete` method at position 12. An `Invoke` action may or may not create a new object as this depends on the method it represents. All other `Action` instances always create a new object which may be referred to by other `Action` instances using their index in the corresponding `ActionSequence`. That is, the nth element of an `ActionSequence` is only valid if the previous $n - 1$ `Action` instances in the corresponding sequence could be executed without a failure.

For the use in the IBPC, the test sets created in this thesis need to be exchangeable. As this thesis focuses on complex data structures, any `ActionSequence` created by the test generator is restricted to contain one and only one instance of the complex data structures which should be tested. In addition, that data structure must always be initialized as early as possible in the sequence. Doing so, the generated sequences may easily be exchanged throughout the different

Table 5.2: Representation of the method sequence example (Listing 1) as used by the test set generation framework in PTAF.

position	type	method	object	result type	values
0	`Initialize`	`init`	-	`RTree`	-
1	`Primitive`	-	-	`double`	*0.0*
2	`Primitive`	-	-	`double`	*1.0*
3	`Initialize`	`init`	-	`NDPoint`	1, 1
4	`Initialize`	`init`	-	`NDPoint`	2, 2
5	`Initialize`	`init`	-	`NDRectangle`	3, 4
6	`Initialize`	`init`	-	`Object`	-
7	`Invoke`	`getReference`	(static)	`ObjectReference`	6
8	`Initialize`	`init`	-	`NDRectangleKey`	5, 7
9	`Invoke`	`insert`	0	`boolean`	8, 7
10	`Initialize`	`init`	-	`DefaultRectangleQuery`	
11	`Invoke`	`getContained`	0	-	5, 10
12	`Invoke`	`delete`	0	`boolean`	8

data structures by simply exchanging the first initialization of the data structure with another one. As all structures should support the same interface, they are fully compatible.

5.2.2.2 Selection

For the selection of the population used for one cycle of a genetic algorithm, the algorithms already implemented in the MOEA framework are used. The choice of the selection operator strongly depends on the algorithm and its general design, for instance whether it is a single objective algorithm or a many objective algorithm. The chosen selection operator will be defined in the evaluation chapter (Chapter 6) together with the chosen genetic algorithms.

5.2.2.3 Recombination

The recombination in a genetic algorithm chooses one or more parent solutions and rearranges them to one or more child solutions. For the special case of method sequences which are desired here, recombining different solutions with one another or even rearranging a single solution is very error-prone. The atomic parts of one solution are strongly connected to each other. Any known recombination, such as a crossover or a simple shuffling would much likely cause a failing sequence. Therefore, only a single-point crossover is applied on complete test sets, i.e. the `ActionSequence` instances between two `TestSet` instances may be exchanged with a given probability. For instance, considering two test sets with n and m `ActionSequence` instances, the first k elements in the first test set are swapped with the first k elements in the second test set, $k \in \{0, ..., \min(\text{length}(n), \text{length}(m))\}$.

5.2.2.4 Mutation

A mutation changes an existing solution creating a new one. The mutation operations described here are newly designed for the purpose of this thesis. The test generator described

here mutates a test set on three levels: The mutation of test sets, the mutation of action sequences and the mutation of actions.

First, the mutation operator mutates the test sets. A test set is incremented or decremented, i.e. new empty sequences are added to a test set or existing ones are removed from the test set. Secondly, an action sequence may be mutated. As this faces the same problems as the recombination - the high possibility of an invalid sequence - only the incrementation and decrementation of existing sequences are applied. Extending a sequence or removing the last action in a sequence will not cause an invalid sequence.

Existing test generators rely on the diversity of the genetic algorithm to produce longer and valid sequences whose parts rely on each other. For instance, consider the example of a valid sequence in Listing 1. Starting with the `double` each following action has to be added by the genetic algorithm randomly, guided by a rather complex fitness function. PTAF does not use this *bottom-up* approach but a *top-down* approach, i.e. first the desired method is chosen and then all necessary parameters created. Algorithm 5 shows how a sequence is incremented by a new action.

```
1  addNewAction (required action a, sequence s, interface i)
2      if a is a primitive then
3          create primitive with a new random value of the desired type and add it to s;
4      else if a is an array then
5          create array of random length l;
6          for k = 0, ..., l − 1 do
7              add createParameter(basic type of a,s,i) to a;
8          end
9          add a to s;
10     else
11         foreach parameter p in a do
12             add createParameter(type of p,s,i) to a;
13         end
14         add a to s;
15     end
16 end
17 createParameter (required type t, sequence s, interface i) : index of created parameter
18     if s contains actions which create objects of type t then
19         return getRandomObject(s,t);
20     else
21         action a =getRandomAction(i,t);
22         addNewAction(a,s,i);
23         return index of recently created action;
24     end
25 end
```

Algorithm 5: Algorithm to increment a sequence by adding a new action. The type of the action is randomly chosen with incrementation from all available actions given by the interface.

When incrementing a sequence, an action is randomly chosen from the set of available actions defined by the given interface. The parameters required by this action are chosen randomly and recursively. That is, with a certain probability, if values of the required type already have been created in the corresponding sequence, one of them is randomly chosen as parameter. If not, a new value of the required type is created recursively. Therefore, a random action from all actions in the interface which return a value of the required type is chosen to create the desired value. The values for the creation of this random action are then chosen recursively. Again, either already created ones are used at random or new ones are created. Primitive values and arrays do not need to be added to the interface; they are a part of the default set of actions in PTAF. The base case of the recursion are primitives and arrays of primitives or of types which are defined by the given interface, as well as existing values if the algorithm randomly choses to go with already existing values. If the base case of the recursion is reached, the actions are added to the sequence starting with the action created in the base case and continuing to the first action that has been chosen with the incrementation of the sequence.

For instance, if the genetic algorithm chooses to create an invocation of a `max(int,int)` method, it first needs to identify the two `int` values which should be used. The algorithm would randomly decide if already existing `int` values should be used or new ones created. If it is decided to create new values, all methods in the interface are identified which return `int` values. Then either one of these methods is used or the primitive values are simply initialized. This is decided randomly. If a method is chosen to create the values, its invocation and the creation of the required parameters is again built up recursively. If it is decided to simply initialize the primitives, a `Primitive` instance is added to the action sequence. In the end, the `Invoke` instance which represents the invocation of the `max(int,int)` method, is added to the action sequence, referring to the position of the corresponding actions in the action sequence.

The third level of mutation is the mutation of the single actions of a sequence. Possible changes on single actions are:

- Changing the value of a primitive.
- Changing the length of an array or permuting the elements in an array.
- Choosing other values for a method or constructor by randomly choosing other values from the pool of existing values of the required type in the corresponding sequence.

5.2.2.5 Fitness Function

The heart of any genetic algorithm is an appropriate fitness function. It needs to direct the selection of the child population such that mutations are chosen which lead to the best possible solutions. That is, the fitness function ideally needs to be a continuous function such that the optimum of the search space is reached in a series of optimization steps with an increasing fitness. The branch distance function for instance, which is explained in Section 2.11.4.1, is designed to continuously target branches which are only reached if a certain boolean condition is met. For the IBPC implementation discussed here, the fitness function needs to direct the genetic algorithms to generate test sets with a high coverage. For the special case of high-dimensional spatio-temporal index structures, which is the focus of this thesis, not only test sets with a high coverage but tests which reflect the impact of increasing dimensionality are

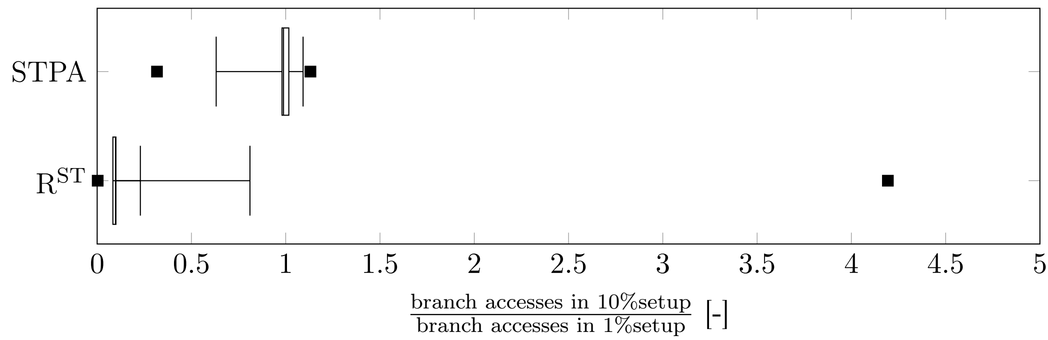


Figure 5.7: Relative branch accesses. Box Whisker chart over the number of branch accesses of the 10% setup divided by the number of branch accesses of the 1% setup. The dots mark the minima and maxima. Branches with zero accesses in either setup are ignored.

needed. In order to create such a fitness function, the classic workloads used in Section 3.5.1 are analyzed with respect to their coverage. A fitness function is created which is able to direct the genetic algorithms to create test sets which reflect the capabilities of the classic workloads. This is done by answering several research questions which are described and answered below. In the end, the final question on how the fitness function should look like, is answered.

Are there differences in the coverage of workloads with a different number of elements? The computation of coverage is very expensive in terms of computation time. Especially the growth of invocations during the computation of the coverage of the R^{ST}-tree makes it nearly impossible to generate results over the complete set of workloads with an appropriate number of iterations (≥ 100). Therefore, the number of elements in an index is reduced to 10% and 1% of the original workload which only requires three weeks of computation time on a Dell Precision M4800 CTO with 16GB 1,600 Mhz DDR3L RAM, Intel Core i7-4810MQ and a Samsung 520/540 850 EVO Basic SSD. For both setups, the covered branches of all workloads are recorded. Then the difference between the two setups is computed. Each branch which is only covered by one of the setups but not the other is recorded. The result is that no such branch exists for the R^{ST}-tree. Strictly speaking, on the branch coverage level, both setups cover exactly the same code. For the STPA, three branches are recorded which are covered by the 1% but not the 10% setup. That indicates that above a certain value the total number of elements in an index does not influence the branch coverage at all.

As this thesis focuses on the performance of the workloads, also the differences in the total number of branch accesses are recorded. With ten times more elements in one than in the other setup, there should be differences in the total number of accesses. Therefore, the relative number of accesses is recorded, i.e. the ratio of accesses on a branch between the 10% and 1% setup. Figure 5.7 shows the relation between the total number of branch accesses for all branches and over all workloads for the 10% and 1% setup. The points denote the minima and maxima. For most of the branches, the relation is close to 1 for both, the STPA and the R^{ST}-tree. That is, these branches are not only covered but also accessed nearly the same number of times for either number of elements in the indices. The highest ratio recorded is 4.19355 for one branch in the R^{ST}-tree. Naturally, the ratio between the number of accesses should not grow linearly with the number of elements as also the execution costs of the indices

is not linearly (see Section 3.5.2) growing. Nonetheless, the research question on the impact of the number of elements on the coverage can be answered. There does not seem to be an impact on the coverage of the STPA and the RST-tree if one workload uses more elements than the other. As conclusion, not the original workloads but the workloads with a reduced total number of elements are used for the further analysis of the coverage.

What are the differences in the coverage of workloads with different distributions? In Section 3.5.2, the performance of the STPA and RST-tree clearly show differences when measured for different spatial and temporal distributions of the data. In order to generate a test generator which is able to create test sets that reflect this behavior, it needs to be investigated how the different distributions affect the coverage of the workloads. It needs to be determined which type of coverage metric is able to reflect the effect of different distributions on the performance. The workloads are separated by the used distribution. As a reminder, three different distributions are used by the classic workload generator: a uniform, a gaussian and a skewed gaussian distribution (Section 3.5.1). The three groups of workloads are compared pairwise identifying those branches which are covered by the workloads with one but not the other distribution. This is displayed in Table 5.3.

Table 5.3: Total number of branches covered by the distribution named on the column and not by the distribution named on the row.

(a) RST-tree (1298 branches in total)

distribution	uniform	gaussian	skewed
uniform	0	10	14
gaussian	0	0	4
skewed	0	0	0

(b) STPA (912 branches in total)

distribution	uniform	gaussian	skewed
uniform	0	2	4
gaussian	4	0	2
skewed	4	0	0

The results indicate that there is an influence of the distribution of the elements used in a workload on the branch coverage. This effect is greater for the RST-tree as more branches exist in total, i.e. the sum of all elements in one table, which are not covered by the workloads of all distributions (a total of 28 branches for the RST-tree and a total of 16 branches for the STPA). The results in Section 3.5.2 support this assumption. The skewed distribution seems to have the greatest impact on the query performance (Figures 3.9 and 3.10) and Table 5.3 shows that the skewed distribution also has the greatest impact on the coverage differences. Those branches that are covered by the workloads of one and not covered by the workloads of another distribution do not share a common characteristic. They are neither part of a recursive function nor part of a single method that is totally covered/not covered nor does the condition that needs to be fulfilled to reach those branches correlate in any way. This means for the test sets to be created that there needs to be a great variance in the distribution of automatically created and inserted elements in the index.

Which branches in the RST-tree are correlated to the *curse of dimensionality*? The major conclusion in Section 3.5.2 is that the RST-tree is clearly affected by the curse of dimensionality but the STPA is not. The fitness function used by the test generator of PTAF should be able to reflect that effect in the test sets it creates. In order to direct the genetic

algorithms to create such test sets, it is searched for the indicators for a worsening performance in the coverage.

Therefore, all existing workloads are grouped by the number of dimensions of the generated data. As the workloads for data sets of seven numbers of dimensions $(5, 10, 15, 20, 30, 40, 50)$ exist, seven groups of workloads exist. In each of these groups, the mean of the query performance of the in-memory case (Section 3.5.2) is computed. Ordered by the number of dimensions, this list of mean performance values shows linear behavior. In order to detect metrics which reflect this performance behavior, different metrics are correlated to the list of seven performance values. Since the mean performance shows linear behavior with increasing dimensions, both, the Spearman and the Pearson correlation coefficients are computed.

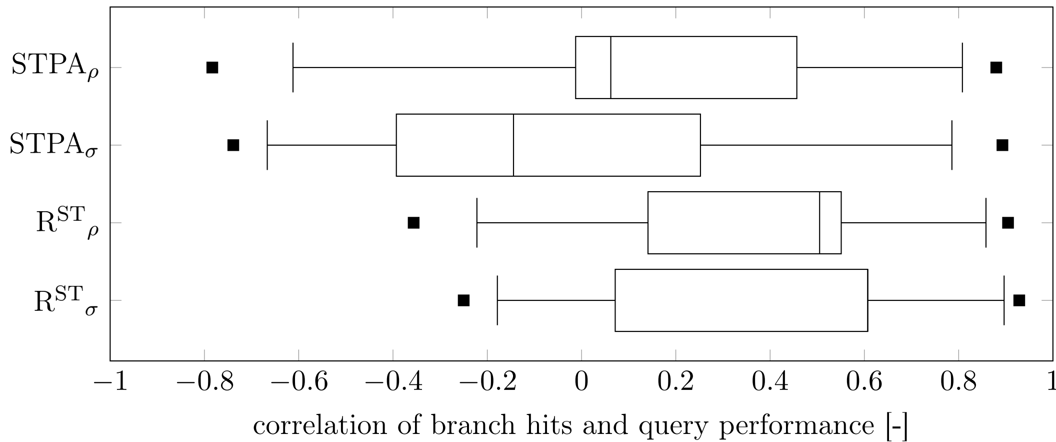


Figure 5.8: Spearman (σ) and Pearson (ρ) correlation for the total number of branch hits to the query performance for every branch. The points represent the minimum and maximum correlation. All branches that are not covered are excluded from the computation.

Figure 5.8 shows the correlation of the branch hits with the query performance for every branch in the STPA and the R^{ST}-tree. That is, for all workloads in each of the seven groups, the hits on each branch are summed up. This results in seven groups, each containing n sums, with n being the number of branches in the corresponding index. The number of hits of each branch is then correlated to the query performance, resulting in one correlation value for each branch, i.e. n correlation values. The distribution of the correlation over all branches is then shown by a box plot.

More branches in the R^{ST}-tree show a higher correlation than in the STPA. The maximum correlation in the R^{ST}-tree is 0.928571 for Spearman and 0.904759 for Pearson correlation. In addition, the minimum correlation for the STPA is much lower than for the R^{ST}-tree. Overall, the curse of dimensionality does not seem to influence all branches in the R^{ST}-tree. Moreover, the difference to the distribution of correlating branches in the STPA does indicate the effect of the curse of dimensionality on the R^{ST}-tree. The branches which show the highest correlation to the performance are examined individually. For both, the R^{ST}-tree and the STPA, these branches are part of loops which run over all dimensions of the data space.

The correlation behavior of loops is shown in Figure 5.9. It is computed like the branch hit correlation. For each of the seven groups, each representing a certain number of dimensions in the workloads, the number of loop iterations on the different loops is summed up. This results

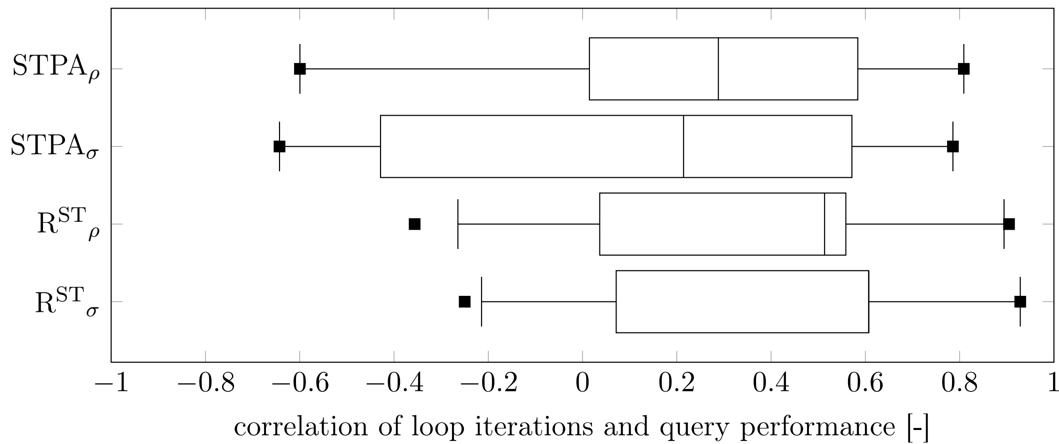


Figure 5.9: Spearman (σ) and Pearson (ρ) correlation for the total number of loop iterations to the query performance for every loop. The points represent the minimum and maximum correlation. All loops that are not covered are excluded from the computation.

in seven groups, each containing n sums with n being the number of loops. Each loop is then correlated to the query performance, resulting in n correlation values. Strictly speaking, it shows the correlation of the total number of hits on those branches which are back edges of a loop to the query performance. These branches show similar behavior as the correlation of all branches. The correlation of branches of the STPA seems to be higher if only back edges are considered. This might result from the fact that most of the loops in the STPA are bounded by the number of dimensions. But no additional conclusion can be drawn if the chosen branches are only back edges. The reverse conclusion, that the limitation on back edges does not exclude informations which may be made when considering all branches, should be noted.

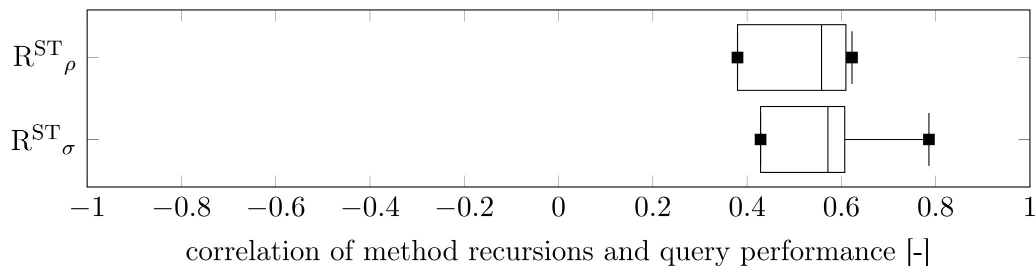


Figure 5.10: Spearman (σ) and Pearson (ρ) correlation for the total number of recursive method calls to the query performance for every method. The points represent the minimum and maximum correlation. All methods that are not covered are excluded from the computation.

Like branches and especially loops, recursive method invocations are examined, too. Figure 5.10 shows the correlation of the total number of recursive method invocations to the query performance for each recursive method. A recursive method invocation is recorded when a method is invoked again before the invoked method reaches one of its end nodes. That is,

method recursions are not only considered to result from a method invocation inside the very same method but also from invocations from a method that has been invoked by the recursive method. The STPA does not contain a method that has been invoked recursively through the execution of the workloads. In contrast to the loops, the correlation of all recursive methods to the query performance is positive. With the median lying around 0.6 for both, the Spearman and Pearson correlation, the correlation is not really strong but it should be noted that the minima are only around 0.4. That is, a correlation can not be excluded for any of the recursive methods.

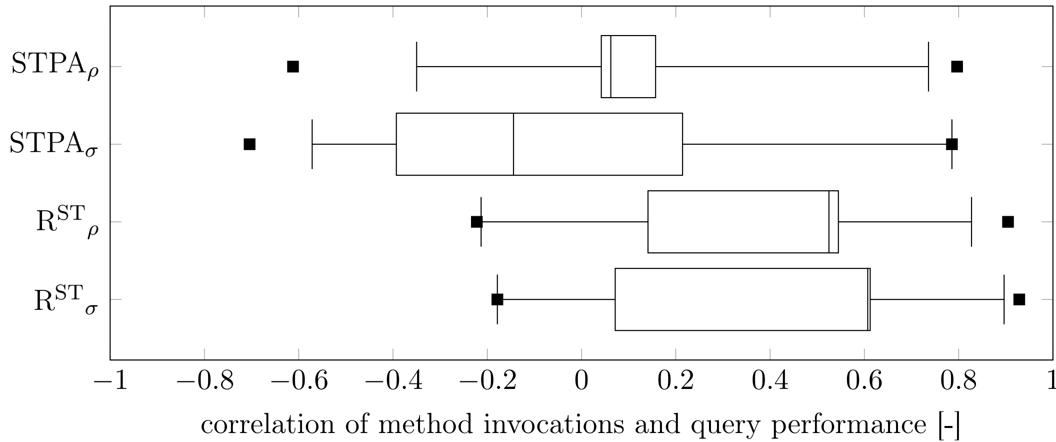


Figure 5.11: Spearman (σ) and Pearson (ρ) correlation for the total number of method invocations to the query performance for every method. The points represent the minimum and maximum correlation. All methods that are not covered are excluded from the computation.

At last, also the correlation of the total number of invocations of each method to the query performance is computed which is shown in Figure 5.11. The correlation of the method invocations shows a behavior similar to the correlation of the total number of branch hits. The method hits seem to describe a part of the effects recorded for the branch hits.

What should the desired fitness function look like? The coverage of classic workloads on the STPA and RST-tree is examined in three directions in the previous paragraphs. First, the effect of a reduced total number of elements in an index is investigated. The analysis shows that setups with a lower number of elements in total may be used for the analysis as good as the setups with a higher number of elements, saving computation time for the computation of the coverage. Secondly, the effect of different distributions on the coverage is measurable and therefore, the test generator should tend to create test sets with a high diversity in the inserted elements. Thirdly, the employed metrics indicate that there is no significant correlation of the curse of dimensionality to the coverage. Nonetheless, high correlations can be found when examining the branch hits, loop iterations, recursive method invocations and all method invocations. In addition, the behavior of the branch hits is similar to the behavior of the loop iterations and method invocations.

For the fitness function, Section 2.11 indicates that many objective genetic algorithms are the most promising approach. As none of the existing test generators is able to generate test sets based on an interface with a high coverage on the given set of R-tree implementations

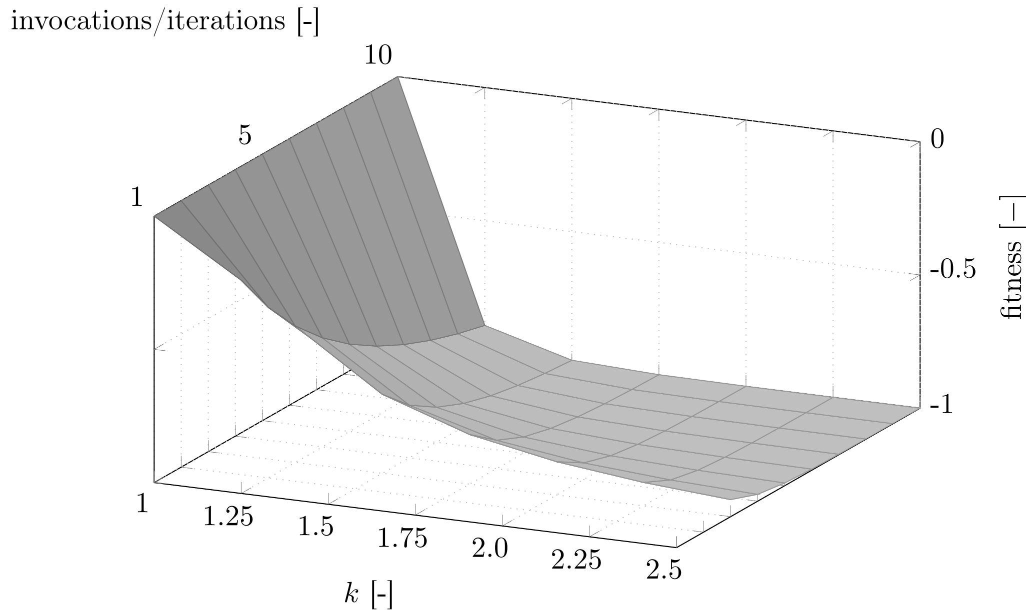


Figure 5.12: Behavior of the fitness function for targeting a back edge or a recursive method.

(Section 5.2.1), a new function is required. Like the most effective approach on unit test generation DynaMOSA [149], this new function should also target many objectives, i.e. it should target each objective individually instead of combining all possibly weighted objectives. In order to save computation time and reduce the number of objectives and to avoid the typical problems of a great number of objectives (see Section 2.11.4.1) not each branch is targeted individually but only those which are back edges. In order to not miss any branch, also the overall branch coverage is added as objective. Menninghaus et al. (2017) [135] show that adding the coverage on all methods as optimization goal increases the overall coverage because this increases the possibility that methods with a low number of branches are targeted, too. The number of loop iterations and recursive method invocations should be maximized in order to incorporate test sets with a poor performance, i.e. with a high computation time. Both target types are treated the same way and each back edge and each recursive method is added as an objective. Nonetheless, maximizing the number of loop iterations and recursive methods could lead to test sets which only concentrate on loops and recursive methods. Therefore, the number of loop iterations and recursive method invocations should be maximized until a certain user defined level of satisfaction is reached. This allows the user to define how important a high number of loop iterations and recursive method invocations is. The following function is used to incorporate recursive method invocations and loop iterations as objective into the many objective fitness function:

$$f_i(n) = \frac{1}{k^n} - 1 \tag{5.1}$$

Here, n is the total number of recursive invocations and loop iterations, respectively. k is a constant which denotes how many invocations/iterations are needed to achieve a full “coverage”. The -1 is added since all objectives in the fitness function are to be minimized. Figure 5.12 shows how k and n influence the fitness function.

Summing up, a new many-objective fitness function is used for the automatic generation of performance tests. The objectives are the overall branch and method coverage of the SUT in $[0, 1]$, the branch coverage of each method in $[0, 1]$ and the coverage on loop iterations and method recursions in $[-1, 0]$ as defined in Equation 5.1. As the method and branch coverage should be maximized and the fitness function usually minimizes all objectives, the coverage values are simply inverted. This function should employ the diversity in test sets as required by the analysis in the previous paragraphs. Employing the fitness function of method recursions and loop iterations benefits the generation of performance tests, e.g. when creating performance tests to unfold the curse of dimensionality.

5.2.2.6 Implementation

The IBTG is implemented on top of the multi objective evolutionary algorithm (MOEA) framework [96]. Since a specialized representation of the test sets is used instead of a binary representation, each operator for recombination and mutation has to be implemented anew. The main benefit of using the MOEA framework is that all already implemented genetic algorithms and selection operators may be re-used. The implementation is evaluated in more detail in Chapter 6.

5.2.3 Optimizing the Length of Test Sets

When the IBTG ends its computation due to the fact that each objective reaches its maximum fitness or the given resources, e.g. the overall computation time, are spent, this results in a set of test sets, each consisting of action sequences. After computation, all test sets are merged to a single test set. Each action sequence of this test set is minimized as shown in Algorithm 6 in order to reduce overhead computation when executing the test sets. Each action that is not needed to execute another action and whose deletion from the action sequence does not reduce the overall fitness is removed from the action sequence. Analogously, all action sequences from the test sets are removed which can be removed without worsening the overall fitness.

Input : action sequence as a list a of n actions a_i
Output: a minimized action sequence

```
1  for i = n − 1, i ≥ 0, i = i − 1 do
2      if !IsRequired(a_i, a) then
           /* a_i is not required by an a_j in a with j > i    */
3          f_0 = ComputeFitness(a);
4          f_i = ComputeFitness(a \ a_i);
5          if f_0 ≤ f_i for each objective then
6              delete a_i from a and update i and n
7          end
8      end
9  end
10 return minimized action sequence a;
```

Algorithm 6: Algorithm to minimize the number of actions in an action sequence a without worsening its fitness.

5.3 Alternative Test Set Generators

The interface based test generator (IBTG) described in Section 5.2.2 is designed in a very abstract way in order to not only support the generation of test sets for the comparison of index structures but all other types of systems. As this may lead to non-optimal test sets, two other more specialized test generators are created and described in this section: The *guided workload generator* optimizes the parameters of the classic workload generator (Section 3.5.1) by a genetic algorithm and the *specialized test generator* generates test sets similar to the test generator described in the previous section but with the knowledge of the specialities that are needed for spatial and high-dimensional spatio-temporal indices. The following subsections describe the differences between the generators and their major implementation details. Figure 5.13 gives an overview of the workload and test generators used for the remainder of this thesis. The guided workload generator inherits from the classic workload generator (Section 3.5.1), the specialized test generator inherits from the IBTG described above. The classic workload generator is the most specialized generator and only depends on the correct configuration by user experience. The IBTG is the most general approach and only requires a valid interface to create the desired test sets. All generators are evaluated in Chapter 6.

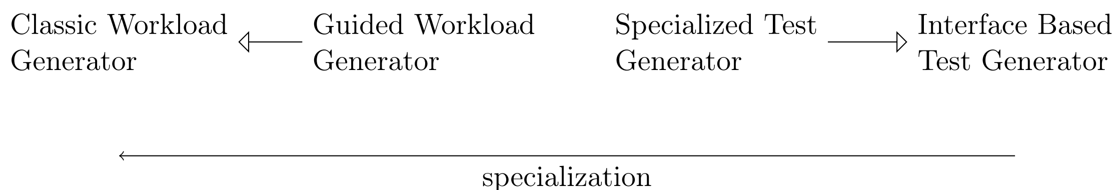


Figure 5.13: Overview of the four different test generators used in this thesis. Two workload generators (left), and two generators for test sequences (right). Whereas the classic workload generator is completely dependent on user experience, the automated test generator only requires a valid test interface and automatically creates appropriate test sets.

5.3.1 Guided Workload Generation

The parameters for the classic workload generator (see Section 3.5.1) are set with respect to the user experience on the indices to test and the requirements of these indices. Since the parameter space is too large to be explored exhaustively, a genetic algorithm is used to generate a setup for the classic workload (see Section 3.5.1) that is designed to achieve a high coverage on the code. It uses the very same fitness function as described in Section 5.2.2.5 for the IBTG. The algorithm not only searches for one workload setup but for a set of setups which leads to the best fitness. Despite the optimization of the parameters by a genetic algorithm, two other adaptions to the classic workload generator are made by introducing a new parameter. The `sizesCount` denotes how many steps a workload has. As a reminder, the classic workload generator always inserts, updates or deletes a total of `incSize` histories 10 times. Thus, the classic workload generator would have a `sizesCount` of 10.

The genetic algorithm varies every parameter of the classic workload generator. In each generation, a parameter value is varied when a certain user-defined probability is met. When a parameter is varied, this is done with respect to the range and the operator types listed in Table 5.4. The parameter ranges are chosen with respect to the outcome of the evaluation in

Section 5.2.2.5 and the constraints of the overall application. For instance, a `startPercentage` outside $[0, 1]$ will always cause a corrupted workload as the sum of `startPercentage`, `endPercentage` and `updatedPercentage` must be ≤ 1.0. A parameter value may be varied by any of the listed operators in any order. For instance, the initial number of elements in an index `initialSize` may be varied by the half-uniform crossover *HUX* operator first and then by the *BitFlip* operator. Each operator chooses the parent solutions by using the selection operator of the specific genetic algorithm. The operators are chosen to reflect the two basic operators of a genetic algorithm: the crossover of two values and their recombination and the mutation of a single value (Section 2.11.4.1). All used non-custom operators are already implemented in the MOEA framework [96].

Table 5.4: Parameters varied by the guided workload generator, their ranges and the operators used to vary them.
*: the parameter is not a part of the classic generator
**: custom parameter type to represent a distribution

parameter	type	range	variation operator
`startPercentage`	floating	$[0.0, 1.0]$	*PM*, *SBX*
`endPercentage`	floating	$[0.0, 1.0]$	*PM*, *SBX*
`updatePercentage`	floating	$[0.0, 1.0]$	*PM*, *SBX*
`distribution`	distribution**	custom**	custom**
`validTimeDistribution`	distribution**	custom**	custom**
`vtInfinityProbability`	floating	$[0.0, 1.0]$	*PM*, *SBX*
`maxValidTimeLength`	floating	$[0.0, 1.0]$	*PM*, *SBX*
`maxElementSize`	floating	$[0.0, 1.0]$	*PM*, *SBX*
`initialSize`	integer	$[1, 10]$	*BitFlip*, *HUX*
`incSize`	integer	$[0, 10]$	*BitFlip*, *HUX*
`sizesCount`*	integer	$[1, 10]$	*BitFlip*, *HUX*
`queries`	integer	$[0, 10]$	*BitFlip*, *HUX*
`querySize`	floating	$[0.0, 1.0]$	*PM*, *SBX*
`dimensions`	integer	$[2, 50]$	*BitFlip*, *HUX*

The custom operator for the variation of a distribution works as follows: The distribution may be one of the three basic distributions presented for the classic workload generator (Section 3.5.1): uniform, gaussian or skewed gaussian. Making the distributions more adaptable, for instance by not only varying the type of the distributions but the parametrization of the chosen distribution type, leads to a very high amount of failing workloads. Note that each parameter in a distribution is variated on its own, i.e. without the context of the other parameters. Changing one parameter without the other ones will often lead to a set of parameters that is invalid. Therefore, only the discrete change between the three fundamental distribution types is chosen.

5.3.2 Specialized Test Generator

In contrast to the guided workload generator, the specialized test generator does not use classical workload generation to optimize the value of the fitness function defined in Section 5.2.2.5, but a specialization of the IBTG. The IBTG depends on the definition of a specific

test interface in order to create test sets with a high coverage for high-dimensional spatio-temporal index structures. The specialized test generator is specifically designed for that type of structures and does not use a general test interface. Instead, each time a test set or a test sequence needs to be enlarged due to mutation, the enlargement is done with the knowledge of the desired test structure.

Increasing the test set, i.e. adding a new sequence, always means to initialize a new index. As for the automated generator, it is ensured by the specialized generator that one and only one index per test set is initialized and that it is initialized at the very beginning of each sequence. With the knowledge of the index to be initialized, it always can be set up valid in the legal bounds of its parameters, for instance the number of dimensions or the size of its nodes.

Like the classic and the guided workload generator, the specialized test generator saves the histories of the already existing objects for each index. Whenever the genetic algorithm mutates a test sequence such that it has to be enlarged one of the existing histories is updated, deleted or a new history started, each with a probability of $\frac{1}{3}$. All new spatial elements required for the enlargement of a history are created by using a uniform distribution in each dimension. Using the uniform distribution will not prevent the test generator from creating action sequences which add elements with other distributions as the uniform distribution is only the base for the genetic computation. For instance, deletions performed by the genetic algorithm may shift the uniform distribution of the elements to a gaussian or a skewed distribution. Using the uniform distribution only makes it more likely that the added elements are uniformly distributed. If an action sequence is to be decremented via mutation, its elements are only deleted until the actions required for the initialization of the index remain.

All other mutations are set up like for the IBTG (see Section 5.2.2.4). Also, the recombination, selection and the fitness function remain unchanged in comparison to the IBTG.

5.4 Performance Measurement

In order to compare the performance of a set of SUTs which use a common interface, the performance measurement needs to be robust and reproducible. This section describes a new approach for the robust and reproducible measurement of the performance of Java programs.

Georges et al. (2007) [87] compare a large set of performance measurement methodologies and Alghmadi et al. (2016) [30] describe a rather complex measurement pipeline which automatically decides when to stop the performance measurement. As both publications seem to be of the most recent importance for the subject, they are used in this thesis to introduce a new framework for rigorous Java performance measurement. Unfortunately, the framework from Alghmadi et al. (2016) [30] does not seem to be available and all contact attempts in the context of this thesis have not been answered up to the publication of this thesis.

Section 2.6 discusses the main aspects of performance measurement in Java. The greatest impact factor on the measurement is the just-in-time (JIT) compilation. When measuring the performance of a Java program, one always needs to incorporate the uncertainty caused by JIT compilation. Disabling the JIT compilation for performance measurement is not an option. A program should always be tested and executed as it is meant to be tested and executed. In addition, disabling the JIT compiler causes a great computation overhead.

Besides JIT compilation, the performance measurement of a Java program is influenced by scheduling and garbage collection performed by the JVM and by system and user processes

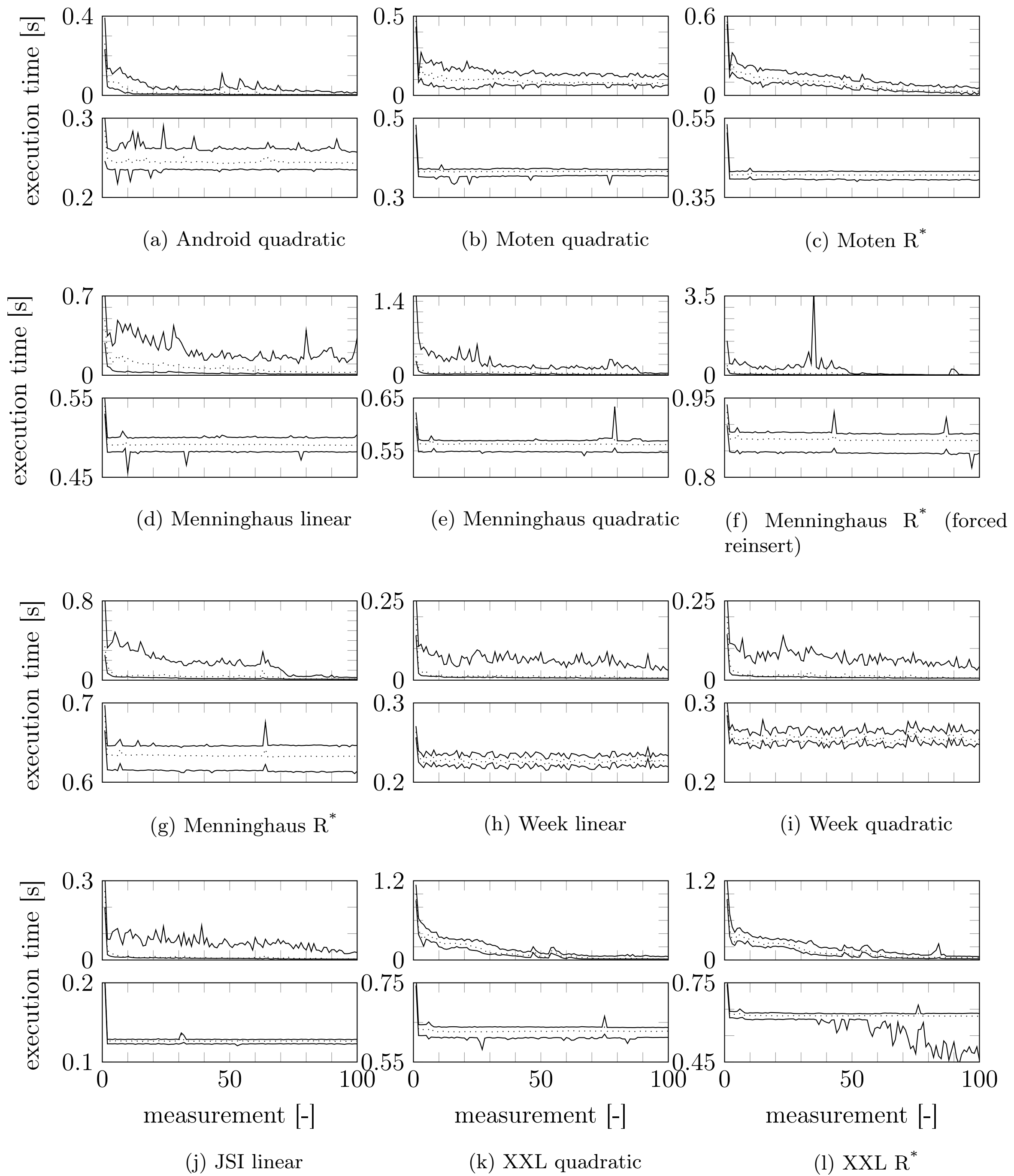


Figure 5.14: Median (dotted) and the 2.5 and 97.5 percentile (solid) of the performance of 100 JVM invocations over the first 100 workload executions with enabled (top) and disabled (bottom) JIT compilation. Note the different scales on the y-axis.

which run in parallel to the JVM. In order to get a glimpse on the magnitude of these influences, the classic workload generator adapted for spatial indices (see Section 5.2.1, page 94) with a

total element size of 100 is computed 25000 times by a single JVM. A total of 100 of these JVM executions is performed for each of the 12 index types described in Section 5.2.1. Another total of 100 JVM executions is performed for each of the 12 spatial index types with disabled JIT compilation and 1000 workload executions per JVM execution. The parameters for each executed workload executed are the very same for each execution. Doing so, the measurements are comparable between the different JVM executions and between any workload execution. All workloads are executed on a Dell PowerEdge R420 with 192GB 1,600Mhz DDR3L RAM, Intel Xeon E5-2420 CPU and four SATA 7,200rpm hard disk drives. Each pseudo random number of a workload bases on a certain random seed making the evaluation reproducible. Boyer (2008) [51] states, that using the `System.nanoTime()` method in Java should be preferred over the `System.currentTimeMillis()` or a `java.lang.management.ThreadMXBean` interface as it is supported by any system, has a higher resolution and a higher reliability. It is also used for the actual performance measurement in this thesis.

Figure 5.14 shows the median and the 2.5 and 97.5 percentile for each setting and each index. Each pair of plots shows the results for activated JIT compilation at the top and the results for the deactivated JIT compilation at the bottom. The median is chosen instead of the mean as the measurements show a very high variability. As each workload consists of numerous method invocations, i.e. the insert, update, delete and query operations, the greatest impact of JIT compilation is expected in the first 100 workload executions of each JVM execution which equals over 10000 basic operations (insert, delete, query) on the indices. Therefore, only the first 100 executions are displayed. As the general behavior is of more interest than the absolute values of the measurements, the scales of the plots are adjusted such that the trends are visible. For each index, the measurement with enabled JIT compilation is very high for the first iterations and then decreases. The measurements seem to be in a steady state for some indices (e.g. Android Quadratic) earlier than for others (e.g. Moten R^*). The measurements for the disabled JIT compilation seem to be steady except for the very beginning. The peak at the beginning most likely results from the time being spend on the warmup of the JVM, for instance the dynamic class loading. Both, the measurements with enabled and with disabled JIT compilation show variations even after they seem to have stabilized. These variations most likely result from the deterministic but somehow uncontrollable garbage collection in Java and from system specific processes, like scheduling and other processes.

5.4.1 Overview

The general assumption behind the new framework is that after a sufficient number of executions of the same test in the same virtual machine, the JIT compilation is finished and no more optimizations will happen. Therefore, the crucial task is to predict whether this steady state has been reached or if more executions have to be performed. If the steady state is reached, a sufficient number of executions has to be computed and the median performance of these executions has to be computed as resulting performance value. The steady state is to be defined by several parameters. As the performance measurements vary even after stabilization or with disabled JIT compilation, it needs to be determined which portion of the measurements needs to lie in which interval in order to be considered steady. Always, only a single JVM execution with the ongoing execution and measurement of the very same test set needs to be monitored as it seems impractical to overwatch several JVM executions and then to decide when to stop each of them. That being said, it needs to be determined how many consecutive executions of the same test set in one JVM execution have to be considered

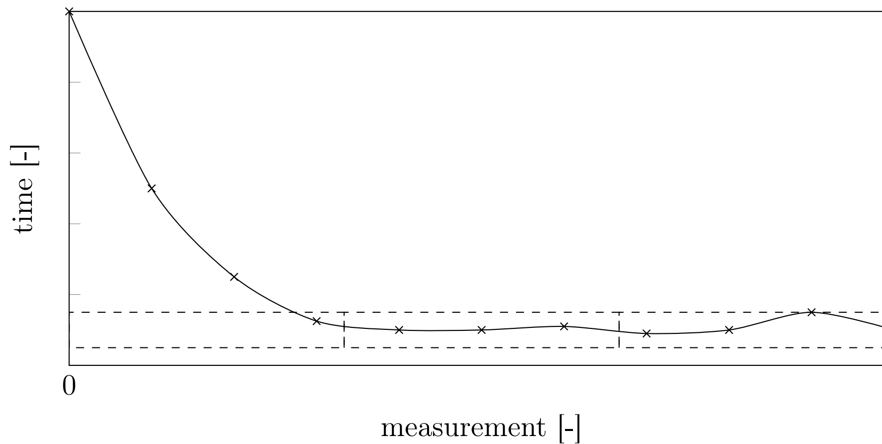


Figure 5.15: Example of the new performance measurement framework for Java programs. The dashed rectangles show how many measurements are taken into account (length on the x-axis) and how big the variations of theses measurements are allowed to be (height on the y-axis) when the steady state should be assumed.

in order to determine a steady measurement. If too many consecutive executions are chosen, the framework needs a lot of unnecessary additional time in order to identify the steady state. If too few consecutive executions are chosen, the framework may identify the steady state too early. The same holds for the interval which is used to identify the steady state. If it is too small, the steady state may never be reached. If it is too great, the steady state may be identified too early. Figure 5.15 shows the general approach. Given an idealized development of performance measurements in the very same JVM, as soon as the variations of the measurement stay inside a predefined range, it is assumed that the steady state has been reached. The height of the dashed rectangles shows how big the variations can be when the steady state should be assumed and the length of the dashed rectangles shows how many measurements are taken into account.

5.4.2 Identification of the Steady State

For a better comparability of the measurements for the different indices and in order to identify the parameters for the identification of the steady state, the measurements are divided by the median of each iteration. That is, the percentiles of the 100 measurements in each iteration are divided by the median of that iteration, with an iteration i being the *i-th* execution of the workload in a single JVM. Doing so, the quantiles of the measurements are determined in relation to the corresponding median. As the percentiles for the steady state should be determined, only the iterations $i = 1.5 \cdot 10^4, ..., 2.0 \cdot 10^4$ are considered for the measurements with enabled JIT compilation. At this point, the influence of the warmup and JIT compilation should be very unlikely. For the measurements with disabled JIT compilations the iterations ($i = 10, ..., 10^3$) are considered. The computed percentiles are shown in Figure 5.16. For instance, the box plot at 0.9 displays the distribution of the 90 percentiles for each iteration over all JVM executions and all indices.

Even under the presumption that the impact of the JIT compilation should be minimal after $1.5 \cdot 10^4$ executions in each of the 100 JVM executions, there are still more variances

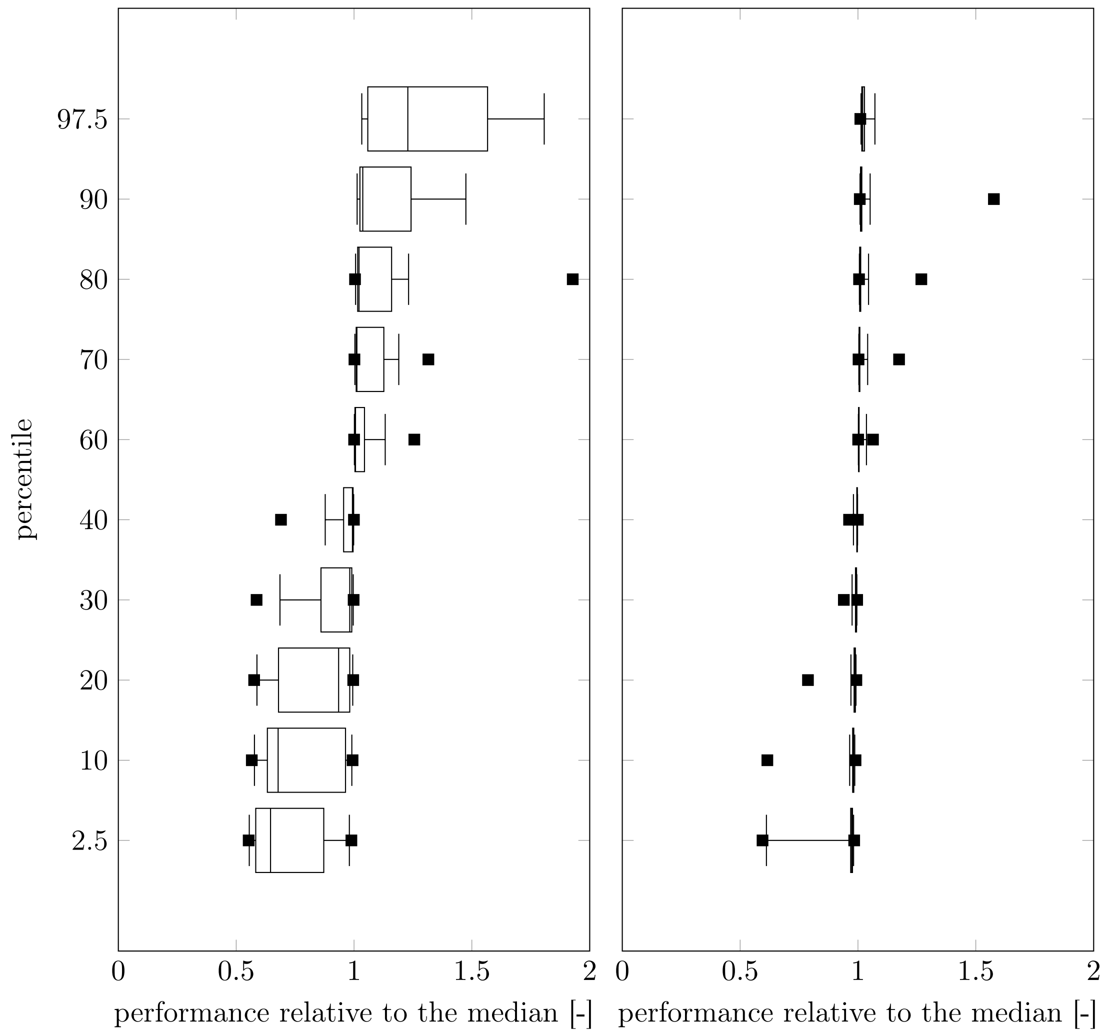


Figure 5.16: Box plots of relative quantiles for the performance measurements with enabled (left) and with disabled (right) JIT compilation over all 12 R-tree implementations.

with enabled than with disabled JIT compilation. One needs to note that the box plots of the percentiles would look very similar even for a higher number of executions. Some of the diversity may also result from the fact that the JIT compilation has a greater impact on one index than on the other indices. That is, the differences between the indices also influence the box plots of the percentiles. This assumption is supported by the different trends of the absolute measurements in Figure 5.14. Nonetheless, all executions need to be considered in order to define an appropriate interval which denotes when the steady state of the measurements has been reached since the test sets which are executed in Chapter 6 may also differ in the impact of JIT compilation. Keeping the described variations in mind, the steady state is defined as to be reached if the measurements between the 2.5 and the 97.5 percentile of the last n executions variate not more than 0.5 around the median. That is, the performance of the last n executions is measured, their median is computed and the n measurements are divided by their median. If the measurements between the 2.5 and the 97.5

percentile lie within the interval $[0.5, 1.5]$, it is assumed that the steady state has been reached. The defined interval is supported by Figure 5.14 assuming that the steady state is reached after more than $1.5 \cdot 10^4$ executions, as only a small portion of measurements lie outside the interval between the 90 and 97.5 percentile. The measurements are cut by the 2.5 and the 97.5 percentile, in order to exclude outliers caused by garbage collection and other processes on the operating system.

5.4.3 Identification of an appropriate number of measurements

The correct number of consecutive measurements in one JVM execution which are used to determine whether the steady state has been reached or not, can only be determined approximately. In order to compare different lengths, the $2.5 \cdot 10^5$ measurements in each JVM execution are partitioned into non overlapping sublists of length n. For each sublist, the median and the 2.5, 25, 75 and 97.5 percentiles are computed and divided by the median. Then the mean values of these percentiles over all sublists at the same position are computed. Doing so, it can be determined how likely it is for a specific length n to identify the steady state and after what number of measurements the steady state is identified. Figure 5.17 shows the described means of percentiles for different lengths of the sublists. The mean percentiles of each sublist are displayed at the end of that sublist. For instance, if the length of the sublist is $n = 1000$, the mean of the first sublist in each execution is displayed at position 1000. In addition, for each sublist, the likelihood of the identification of the steady state is shown.

For a length of 10, there are much bigger variations in the means of the percentiles as well as in the probability for a steady state. In contrast, the variations in the probability for the steady state for a length of 1000 are much smaller. A length of $n = 100$ seems to be a good compromise between an early detection of the steady state and a smaller number of detected false-positives. One has to note that a false-positive is not clearly defined because the steady state can only be assumed. However, a length of $n = 10$ means that a high probability exists to detect the steady state, while the mean of the percentiles (75 and 97.5) is also very high. As this is a kind of a contradiction, for the performance measurement in this thesis, sublists of $n = 100$ are chosen to detect the steady state. The parameter setting described here is given by a rather bounded set of evaluations. It is a good start for a robust and reproducible performance comparison of complex data structures, especially the index structures used for the determination of the parameters. For a use of the new framework for more general Java applications, an even greater and more diverse evaluation has to be made in order to define the parameters more generally. Nonetheless, the given parameters suffice for a robust performance measurement in this thesis. Note that a further evaluation of the performance measurement technique is not useful as the outcome of any evaluation simply reflects the given setup of the steady state.

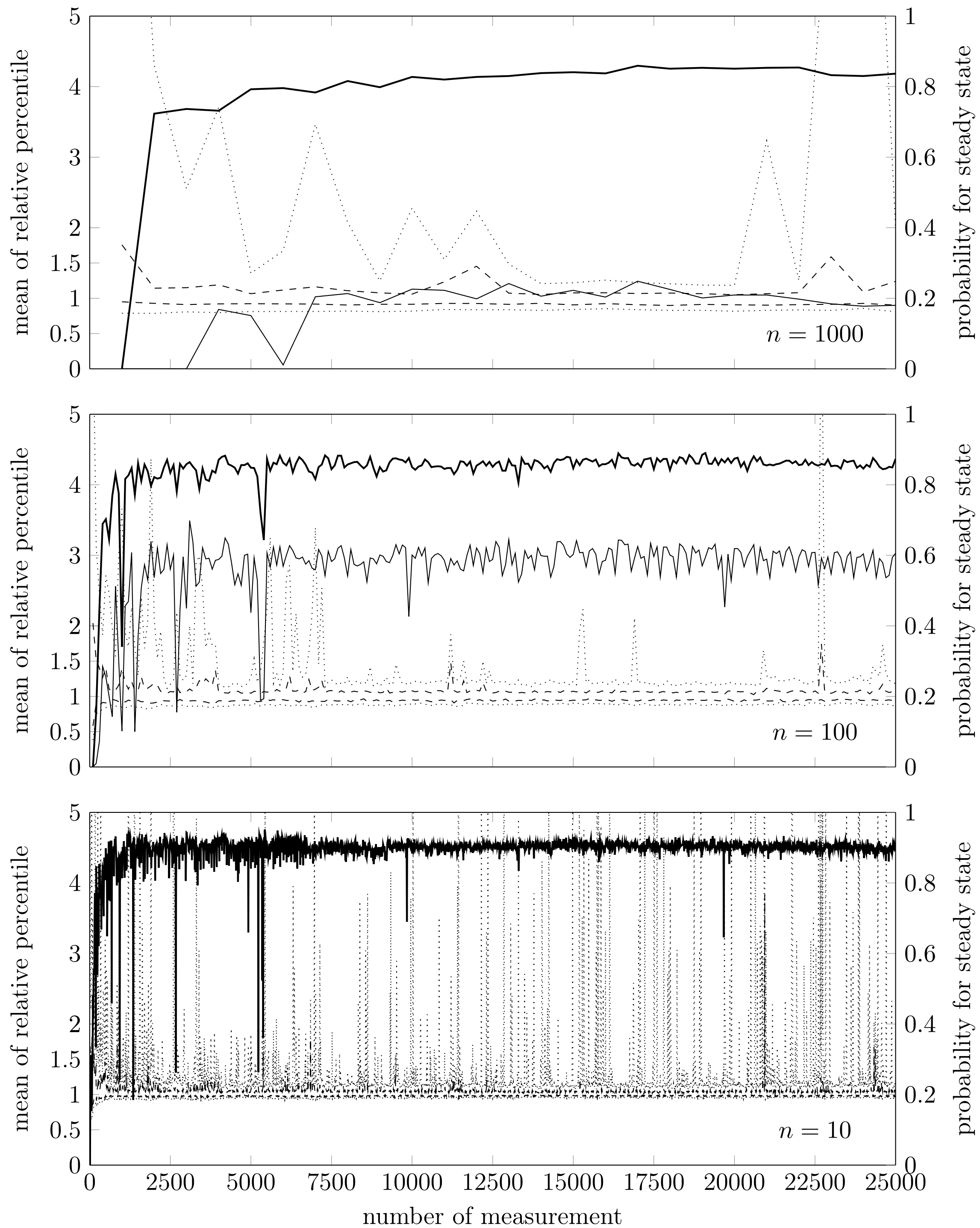


Figure 5.17: Mean of relative 2.5 and 97.5 (dotted), 25 and 75 (dashed) percentiles plus the likelihood of the detection of a steady state if all measurements in the sublist (solid) or only the measurements between the 2.5 and 97.5 (thick) percentile are used.

Chapter 6

Evaluation

This chapter evaluates the Performance Test Automation Framework (PTAF) from several points of view. The branch coverages of the test sets produced by the Interface Based Test Generator (IBTG), the specialized test generator and the guided workload generator (Chapter 5) are compared to the coverage of the classic workload generators for each of the two spatio-temporal (Section 3.5.1) and twelve spatial (Section 5.2.1) indices. With respect to the coverage, several parameter settings for the test generators are evaluated, too. The test sets with the best coverage on the spatio-temporal indices are used for the first case study of the Interface Based Performance Comparison (IBPC) technique. Then the weighted performance (Section 4) resulting from these test sets is analyzed and compared to the general outcome of the evaluation of the classic workload generator (Section 3.5.2). For the second case study, a set of simple data structures is used whose performance behavior is well known.

6.1 Coverage Evaluation for Test and Workload Generators

This section compares the branch coverage of the test and workload generators that are newly created in this thesis. This is done on the basis of the two spatio-temporal and twelve spatial indices described in Chapter 3 and Section 5.2.1, respectively.

6.1.1 Setup

The workload generators are configured as specified in Sections 3.5.1 and 5.2.1. For the computation of the coverage in reasonable time, the total number of elements has to be reduced to 10% of the original workload. Section 5.2.2.5 shows that this does not influence the coverage. Therefore, only the configuration of the other three generators is described: the guided workload generator, the specialized test generator and the interface based test generator (IBTG). All three test generators use the same fitness function (Section 5.2.2.5). In addition, all three test generators use the same set of genetic algorithms and selection operators.

6.1.1.1 Genetic Algorithms and Selection Operators

The genetic algorithms are chosen in order to reflect different approaches in multi- and many-objective optimization using genetic algorithms. Due to the limited number of algorithms used and the specialization of the fitness function, mutation and recombination algorithms, the evaluation may not be used to make general conclusions about the performance of the genetic

algorithms and their selection operators. By using different algorithms, it can be determined whether the structure of the test generation procedure reacts to different algorithms or if the process of optimization does not depend on the selected algorithm. As the MOSA [148] and DynMOSA [149] approach seem to be the most effective approaches for unit test generation (Section 2.11.4.2), their core algorithm, the NSGA-II [75] is also used as genetic algorithm for the generation of performance tests. In addition, two successors of the NSGA-II, the ϵMOEA [76], MOEA/D [120] algorithms are used as they tend to create solutions, which are more diverse and the algorithms work generally faster. The algorithms and their selection operator are listed in Table 6.1. They are all implemented in the MOEA framework [96] as described above.

Table 6.1: The genetic algorithms and their selection operators used for each of the automated test generators.

Genetic algorithm	Source	Selection Algorithm
Random		none
Genetic Algorithm	[102]	Linear Weighted Dominance
NSGA-II	[75]	Tournament Selection with Pareto Dominance Comparator
ϵMOEA	[76]	Tournament Selection with Pareto Dominance Comparator
MOEA/D	[120]	

The *Random* algorithm randomly initializes new test sets in every iteration, i.e. it does not try to enhance the test sets from previous generations. Therefore, the Random approach does not need a selection algorithm. The *Genetic Algorithm* [102] is a simple single objective genetic algorithm as described in Section 2.11.4.1. The fitness values of the objectives in the fitness function are summed up and they are all weighted equally. The *Linear Weighted Dominance* selection operator simply takes the test sets with the best fitness value. The *NSGA-II*, *MOEA/D* and *ϵMOEA* are described on page 43. The *Tournament Selection* algorithm randomly selects a predefined number of test sets from the population. It continuously compares two test sets with each other until the test set with best fitness remains. For each of the selection steps the pareto dominance (Section 2.11.4.2) of the test sets is compared as described by Deb (2000) [72].

6.1.1.2 Bloat Control

Fraser and Arcuri (2011) [81] stress the control of the growth of the length of method sequences, called bloat control (Section 2.11.4.6). Bloat control may be done in different ways in the setup of the test generators. In all three generators, the maximum execution time of a single method sequence in a test set or workload may be limited to a certain value. For the IBTG and the specialized test generator, the length of the test sets may also be minimized by an additional objective in the fitness function. If this objective is added, the fitness function minimizes the length of the method sequences. It should be noted that the length of the test sets is not bounded by a certain limit but simply minimized. In contrast, limiting the execution time of the test sets and workloads implies the definition of a certain limit and therefore it may prohibit test sets which have a disproportional high coverage compared to the length of

the method sequences. In this evaluation, all generators limit the maximum execution time of the workloads and method sequences to 5 seconds each. The analysis of the performance measurement of the classic workload generator in Section 5.4 shows, that none of the classic workloads takes more than ≈ 3.5s to compute. Therefore, 5s should be sufficient to avoid bloating and optimize the overall computation time of the genetic algorithms by not excluding long test sets with a high coverage. The test generation is computed on the very same system as the performance measurement.

6.1.1.3 Desired Number of Loop Iterations and Method Recursions

The fitness function described in Section 5.2.2.5 uses the function $f_i = \frac{1}{k^n} - 1$ to achieve a high number of loop iterations and recursive method invocations n. In the evaluation setup, the parameter k is set to the values 1.0, 1.05 and 1.25 to determine the impact of k.

6.1.1.4 Resource Limit

As the IBTG minimizes the test sets and workloads in different directions, e.g. they try to maximize the coverage and minimize the length of the method sequences, not a perfect solution which fulfills each objective but the best compromise can be expected. Therefore, the overall computation time of the generators is limited. All computations are performed on a Dell PowerEddge R420 with 192GB 1,600Mhz DDR3L RAM, Intel Xeon E5-2420 CPU and four SATA 7,200rpm hard disk drives, the very same that is used for the evaluation of the existing test generators in Section 5.2.1. Each computation is based on a certain random seed. As the evaluation of the existing test generators does not show a difference between a computation time limit of 5 and 30 minutes, all generators in this evaluation are limited to 5 minutes computation time each. Note that the minimization of the method and test sequence described in Section 5.2.3 is performed after the automated generator has finished, i.e. after the desired 5 minutes computation time. Within the minimization all solutions of the pareto optimal sets are combined into one test set for the IBTG and the specialized generator. However, for the guided workload generator, all parameter sets of the pareto optimal set are treated as individual solutions.

6.1.1.5 Test Interface and SUT Definition

The interface based test generator (IBTG) demands a test interface which indicates which methods and initializers can be used to build the test sets. For the spatial indices, the same test interface as presented on page 92 is used. For the spatio-temporal indices, the initializer of the *now*-generator `NowGen(double,double)` is added to the test interface. The first `double` denotes the start time and the second `double` the increment size of the current value of *now* (*now**) per time step. In case of a workload generator, every insert, delete and update operation implies an increment of *now**. In case of the IBTG and the specialized test generator, the increment has to be performed explicitly by invoking the method `incNow()`. The size of the increment value and the number of `incNow()` invocations denote which time span is covered by the *now*-value during the execution of the corresponding method sequence. Therefore, the `incNow()` method is also added to the test interface for the spatio-temporal indices.

The IBTG and the specialized test generator both initialize one index at the very beginning of each method sequence. No index is initialized during the rest of the method sequence and all operations regarding an index are performed on one and only one index structure

per method sequence. All indices that are initialized by the IBTG and the specialized test generator are using a factory method that implements a common interface. All spatial indices are initialized with an implementation of the method `createInstance(int)`, where the `int` denotes the number of dimensions of the index. All spatio-temporal indices are initialized with an implementation of the method `createInstance(int, NowGen)`, where the `int` denotes the number of dimensions of the index and the `NowGen` represents the current value of *now*. In case of the specialized test generator it is ensured that no spatial or spatio-temporal object created after the initialization of the index has another dimension than the dimension of the index. In addition, the number of dimensions is in $\{2, ..., 50\}$. For spatio-temporal data, the specialized test generator ensures that only one `NowGen` is created for each index. In addition, the `NowGen` is initialized within appropriate bounds, i.e. the start value lies in $[0, 1]$ and the increment value is a small fraction of the start value.

All generators need a set of classes to represent the system under test (SUT). The coverage values that are needed to compute the objectives of the fitness function are computed with respect to the complete corresponding SUT. The SUTs are defined by analyzing the source code and identifying those classes which are affected by the methods and initializers of the test interface. The classes from the packages provided by the Java standard library are excluded from the SUTs.

6.1.1.6 Configuration of Mutation and Recombination Operators

The guided workload generator mutates each parameter of the classic workload generator in order to generate a workload with a maximized coverage. The operators are described in detail in Section 5.3.1. The probability for each of the operators to mutate each of the parameters is set to 0.05. Doing so, it is more likely that only one operator is applied on one parameter in a single variation step of the genetic algorithm. The probability for a bit flip is set to 0.1, the default value in the MOEA framework. The distribution index is set to 0.5 which is also the default value in the MOEA framework.

The specialized test generator mutates the test sequences by adding or removing a method invocation or initialization in a method sequence, by adding or removing a method sequence from the test set or by applying a crossover on two test sets. The probability for each mutation and recombination is set to 0.2 which makes it more likely that only one mutation or recombination is applied per iteration of the genetic algorithm. A probability of less than 0.2 would make it too unlikely that the test sets change at all in one iteration.

The IBTG mutates the test sets, the method sequences and the values of the actions as described in Section 5.2.2.4. Each mutation and the crossover of the test sets is applied with a probability of 0.2. As for the guided workload generator and the specialized test generator, the probability is chosen such that for most of the iterations of the genetic algorithm, one and only one mutation is applied on a test set. In contrast to the other generators, the IBTG uses pools of primitive values to randomly generate primitive values. Each primitive pool generates the primitives with respect to a gaussian distribution with a mean value of 0.0 and a standard deviation of 5.0. Doing so, most primitive values that are created lie inside $[-5, 5]$. In order to incorporate constant values such as *not a number* (`NaN`), each primitive pool may generate one of the possible constant values instead of a primitive value. The probability for the creation of a constant value is set to 0.02 and each available constant is chosen with the same probability. These default values are chosen in order to use the IBTG for a larger number of SUTs and test interfaces, not only the spatial and spatio-temporal indices. In future implementations of

the IBTG, the primitive values may be generated based on a pre-evaluation step which, for instance, analyzes the primitive values used in the source code. EVOSUITE [81] already uses such a pre-evaluation step. As most of the results indicate, the chosen parameters suffice for now.

Table 6.2: Setup parameters for the evaluation of the coverage of the different test and workload generators.

parameter	IBTG	specialized	guided
genetic algorithm	Random / Genetic / NSGA-II / ϵMOEA / MOEA/D		
parameter k	1.0 / 1.05 / 1.25		
time limit	5 min		
execution time limit	5 s		
test set minimization	yes / no		-
number of executions	21,000	21,000	10,500

6.1.2 Results and Analysis

This section presents the coverage of the different test sets and workloads that are produced by the three different generators in comparison to the classic workload generator. The main question is whether the automated generators are able to generate test sets and workloads that have a comparable or even higher coverage on the SUTs, i.e. the different indices. In addition, it is shown how the different setup parameters, i.e. the different genetic algorithms, the minimization objective and the control parameter k, influence the coverage of the resulting test sets and workloads. Finally, the results should indicate which configuration has to be chosen in order to generate the test sets for the application of the Interface Based Performance Comparison (IBPC) technique. The setup of parameters is summarized in Table 6.2. Each execution is performed on a Dell PowerEdge R420, with 192GB 1,600Mhz DDR3L RAM, Intel Xeon E5-2420 CPU and four SATA 7,200rpm hard disk drives and the pseudo random numbers are generated on basis of a random seed. The box plots are defined as follows: The lower whisker shows the 2.5 percentile, the upper whisker shows the 97.5 percentile. The box is bounded by the 25 and 75 percentile, the line in the middle denotes the median. The dots denote the minima and maxima.

6.1.2.1 Overview of the Achieved Coverage

Figures 6.1 and 6.2 show the branch coverage of the test sets and workloads generated by the IBTG, the specialized test generator and the guided workload generator in comparison to the classic workload generator. The coverage of all workloads executed by the classic workload generator is combined, i.e. the classic workload generator is represented by a single coverage value. Each of the other generators is executed 50 times and the results for every execution and every configuration are presented in a single box plot per index.

All generators for the R-tree implementations are able to generate workloads and test sets, respectively, which have a branch coverage that is comparable to the branch coverage of the classic workload generator. Compared to the total number of executions only a few workloads and test sets have a higher coverage than the classic workload generator (IBTG:

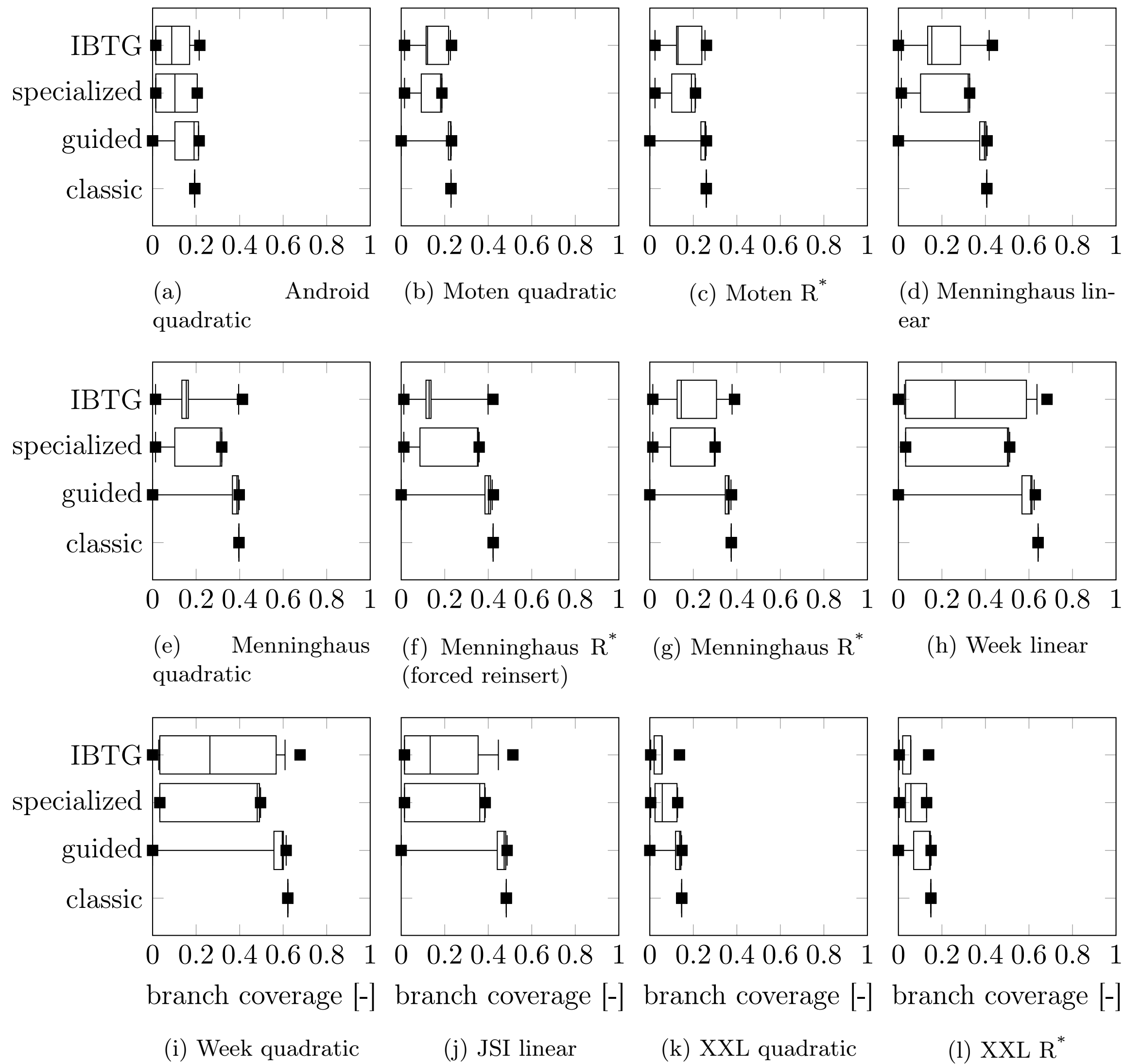
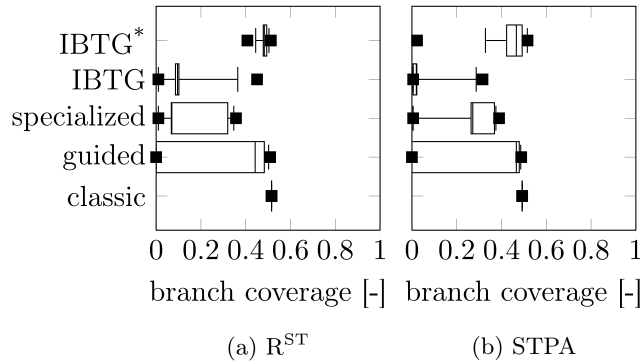


Figure 6.1: Branch coverage of the different workload and test generators for every configuration compared to the branch coverage of the classic workload generator for twelve R-tree implementations.

475, specialized: 693, guided: 548). A high coverage of the classic workload generator generally supports the evaluation of the query performance in Section 3.5.2. Nonetheless, the higher coverage of some executions shows that there is still space for optimization.

For the spatio-temporal indices, the test sets generated with the IBTG and the specialized test generator do not cover the indices as good as the classic workload generator. A detailed analysis unfolds that both fail to configure the `NowGen` and the inserted elements such that the test sets reflect the different temporal relationships, e.g. adding a data set after its valid time has ended etc.. In addition, a lot of the test sets generated by the IBTG fail as the `NowGen` is initialized with non matching or illegal parameters. For instance, the `NowGen` must have a



(a) R^{ST} (b) STPA

Figure 6.2: Branch coverage of the different workload and test generators for every configuration compared to the branch coverage of the classic workload generator for two spatio-temporal indices. The IBTG* denotes the results of an optimized configuration of the primitive object pools and the initial setup.

start value $\in [0, 1]$ and the increment should only be a small fraction of the start value such that the invocation of `incNow` does not increase the current value of *now* above 1.0.

Configuring the IBTG such that only `double` values $\in [0, 1]$ are generated and only one `NowGen` is initialized per method sequence avoids the aforementioned problems. Figure 6.2 shows the results for the ϵMOEA with $k = 1.25$ and no minimization objective as those settings show the overall best results for the default setup. The coverage of the test sets generated by the described configuration shows relatively stable results with a low variation around the coverage of the classic workload generator. The test sets with the highest coverage will be chosen for the prototypal use of the IBPC in Section 6.2.1.

Although the specialized generator provides the most test sets with a coverage higher than the coverage of the classic workload generator, the IBTG provides test sets with a high coverage for each of the implementations. In addition, the IBTG generates the test sets with the overall highest coverage.

For the remainder of this evaluation, the coverage value of each execution is divided by the coverage of the classic workload generator. Thus, the coverage does not need to be analyzed with respect to a specific index and all of these relative coverage values can be combined. In order to get a good overview of the effect of the different configuration, the optimized configuration of the IBTG for the spatio-temporal indices is not included into the further analysis.

6.1.2.2 Analysis of different Genetic Algorithms

Each generator is executed with five different genetic algorithms. In order to detect those algorithms, which provide the test sets and workloads with the highest coverage, the relative coverage values for each algorithm are combined for each generator. The relative coverage is the coverage of a test set or workload divided by the coverage of the classic workload generator for the given index. Figure 6.3 shows the relative coverage of the generators configured with the different algorithms for each of the generators as a box plot.

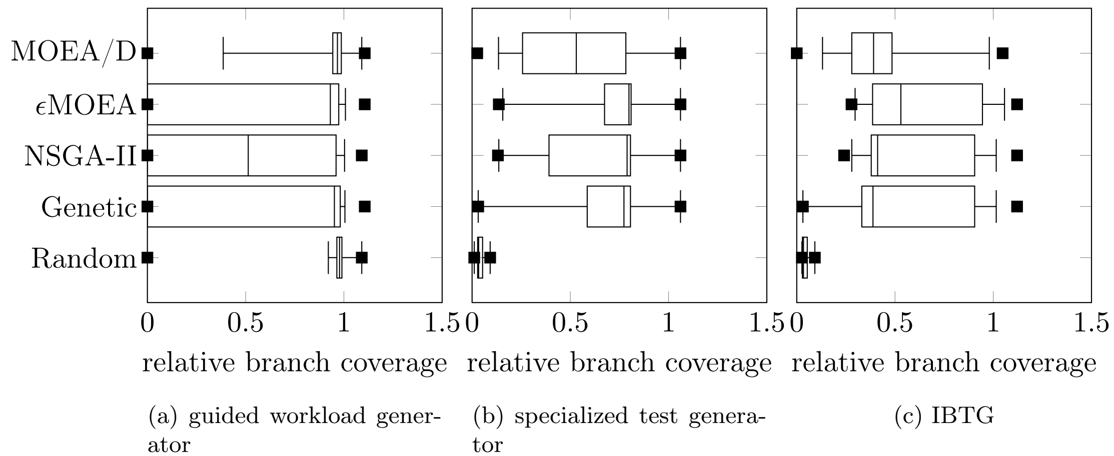


Figure 6.3: Relative branch coverage ordered by the used algorithms over all indices.

The MOEA/D and the Random generator produce the best results for the guided workload generator but the worst results for the specialized generator and the IBTG. This results from the fact that the guided workload generator already starts with a good initial solution which then has to be refined. The IBTG and the specialized generator may only provide good results after several invocations have been added to the initial method sequence. The other three algorithms show similar results for each of the generators, with the ϵMOEA showing the best coverage results.

Note that each algorithm uses the same random seed for the same setup. For instance, the ith execution of configuration c has the same random seed for the MOEA/D as for the NSGA-II. In addition, it should be noted, that the single-objective genetic algorithm, which sums the objectives of the fitness function equally weighted performs comparable to the multi-objective ϵMOEA and NSGA-II.

The evolvability of the genetic algorithms is measured by the change rate (CR) and the population information content (PIC) (Section 2.11.4.5). Since the MOEA/D algorithm splits a multi-objective problem into the optimization of many single-objective problem, the measurement CR and PIC are not applicable. The CR and PIC are only shown for the other three non-random algorithms in Figures 6.4 and 6.5.

The NSGA-II has the highest CRs over all configurations and all indices, the ϵMOEA has the overall lowest CRs and the simple genetic algorithm has CRs that are comparable to the CRs of the ϵMOEA but are slightly higher. The PICs of the simple genetic algorithm are the highest, followed by those of the ϵMOEA and the NSGA-II. The conclusion can be drawn that the NSGA-II has the highest evolvability. Still, the ϵMOEA shows the best coverage results. For future applications it is promising to use the NSGA-II and more time resources.

6.1.2.3 Analysis of the Parameter k

The parameter k is used in the fitness function (Section 5.2.2.5) in order to increase the number of loop iterations and recursive method invocations by a test set or workload setup. In order to avoid a disproportional high growth of loop iterations and recursive method invocations, k bounds the value of the objective for each loop and recursive method $\in [-1, 0]$. Doing so,

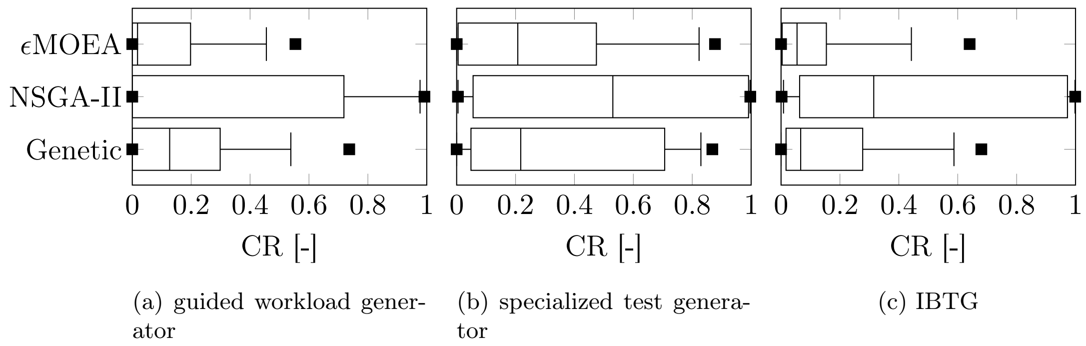


Figure 6.4: Change rate (CR) of the used algorithms over all indices.

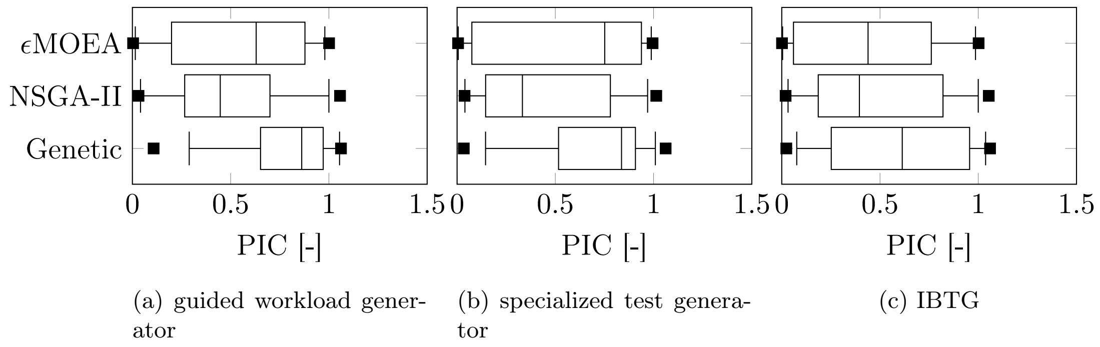


Figure 6.5: Population information content (PIC) of the used algorithms over all indices.

it is more unlikely for the genetic algorithm to only concentrate on more loop iterations and recursive method invocations by ignoring the other objectives.

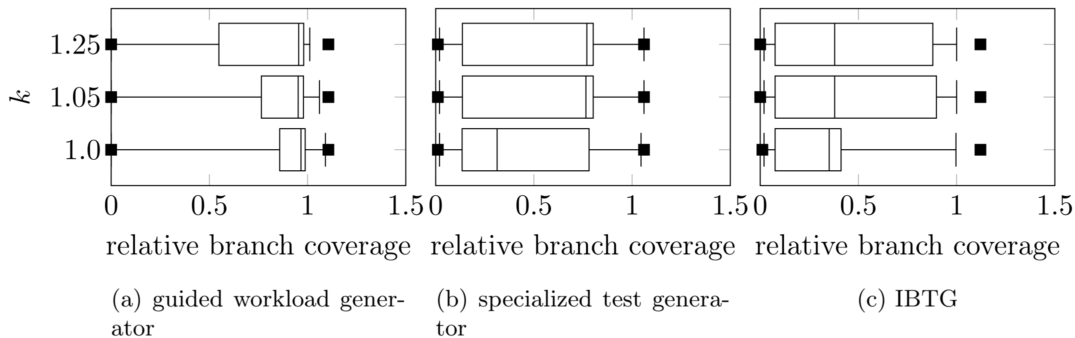


Figure 6.6: Relative branch coverage of different parametrizations of k over all indices.

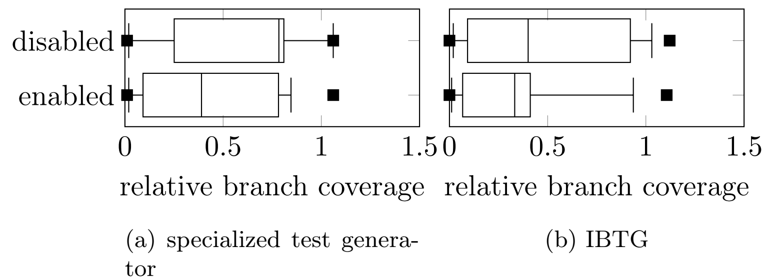
Figure 6.6 shows the relative coverage of the three test and workload generators for each of the three evaluated values of k over all indices. k does not have an influence on the coverage of the workloads generated by the guided workload generator. This most likely results from

the fact that the guided workload generator generates workloads with a lot more elements than the other test generators. More elements always result in more loop iterations and more recursive method invocations. Therefore, the usage of a higher k does not increase the already great number of elements.

For the specialized test generator, $k = 1.0$ generates test sets with a worse median coverage than the test sets that are generated with $k = 1.05$ or $k = 1.25$. A similar observation can be made about the IBTG. Here the difference between a $k = 1.05$ and $k = 1.0$ is even higher. For both test set generators, those setups with $k = 1.25$ generate the test sets with the highest coverage. Nonetheless, all generators are able to generate test sets and workloads with a coverage higher than the classic workload generators. For the generation of performance tests, a higher value of k seems desirable as this forces the test generator to generate test sets with more elements. The evaluation of the IBPC for spatio-temporal indices unfolds the necessity for $k > 1.0$

6.1.2.4 Analysis of Test Set Minimization

The bloat control (Section 2.11.4.6) in the IBTG and the specialized test generator is achieved in two ways: By adding an objective for the minimization of the test set length and by limiting the execution time a single test set. In this evaluation, the latter is chosen with respect to available resources and the expected required maximum execution time of the workloads and test sets. The evaluations of the IBTG and the specialized test generator are all performed with enabled and disabled minimization objective.



(a) specialized test generator

(b) IBTG

Figure 6.7: Relative branch coverage of test sets generated with enabled and with disabled minimization objective over all indices.

Figure 6.7 shows the relative coverage of the IBTG and the specialized test generator for all executions with enabled and with disabled minimization objective. Disabling the minimization objective does not cause bloating of a test set as all test sets are limited by their execution time. All test sets which take longer to execute than the provided 5s get the worst fitness value. Still, the question is wether the minimization of the length of the test sets results in a worse coverage compared to test sets which are generated without minimization objective. For both, the IBTG and the specialized test generator, an enabled minimization objective causes worse coverage results than the disabled minimization objective. As the “hard” bloat control, the limitation of the execution time, seems to be sufficient, the additional minimization of the length of the test sets does not need to be applied.

6.2 Case Studies of the IBPC

This section exemplifies the application of the Interface Based Performance Comparison (IBPC) on two scenarios: The comparison of the performance of the high-dimensional spatio-temporal indices discussed in Chapter 3 and the comparison of the performance of an array list, a linked list and a hash set from the GNU Trove library [23]. The IBPC is designed to compare the overall performance of a set of competitors on the basis of a common interface. Therefore, the application of the IBPC on both scenarios should show which of the competitors has the best performance with respect to the given interface.

6.2.1 High-Dimensional Spatio-Temporal Indices

Section 2.4 shows that no general accepted benchmark for spatial, spatio-temporal and high-dimensional indices exists. Therefore, in Section 3.5.1 the same workload generator as for its competitor, the $\mathrm{R^{ST}}$-tree [172] is used to compare the performance of the spatio-temporal pyramid adapter (STPA). The results of this comparison are compared to the application of the IBPC here. The quality of the performance comparison with the IBPC depends on the coverage of test sets which are generated for each of the competitors. Therefore, the best covering test sets of the optimized test generation in Section 6.1.2.1 are chosen as a basis.

In order to optimize the performance measurement, the length of all test sets is minimized as described in Section 5.2.3 after the test sets have been generated.

The performance of each test set is measured using the performance measurement developed in this thesis (Section 5.4). As the parameters for the detection of the steady state are set by using the evaluation of very similar R-tree like indices, they should suffice for a stable measurement. The mean value of the performance measurements in 100 different Java Virtual Machine (JVM) invocations is taken as the performance of a certain test set executed on one of the indices.

The tables in Table 6.3 show all values computed by the IBPC on the generated test sets. The branch coverage of the test set generated for the $\mathrm{R^{ST}}$-tree has a higher coverage on the STPA than the test set generated for the STPA has on the $\mathrm{R^{ST}}$-tree (a + b). Each test set takes more time when executed on the $\mathrm{R^{ST}}$-tree than executed on the STPA (c). Therefore, for a low weight of the coverage on the weighted performance, the STPA outperforms the $\mathrm{R^{ST}}$-tree. For a high weight of the coverage the $\mathrm{R^{ST}}$-tree outperforms the STPA (d).

Most importantly, this reflects the outcome of the evaluation of the STPA in Section 3.5: The STPA is more efficient than the $\mathrm{R^{ST}}$-tree. It also shows an effect that can not be reflected by the classic approach. The $\mathrm{R^{ST}}$-tree works more stable than the STPA as it is covered by the test sets of the STPA better than its test sets cover the STPA. Generally, the STPA has a greater vulnerability to worst case configurations of the progression of *now* than the $\mathrm{R^{ST}}$-tree. For instance, the query space of the *contained* query needs to be extended for *now*-relative queries (Section 3.3) which may cause many false positive results. The classic workload generator only uses a single progression of *now*. The current value of *now* (now^*) is always beginning at 0.25 and ending and 0.75, regardless of the number of operations. The IBTG generates different progressions of *now*, especially if they are required to reflect different states of the indices. For the $\mathrm{R^{ST}}$-tree, different progressions of *now* are not necessarily required but for the STPA. This explains the differences in the coverage values. For the STPA, it is assumed (Chapter 3) that it can easily be enhanced to a more adaptable index, for instance by using the $\mathrm{P^+}$-tree [212] which is based on the pyramid technique and which is specialized for more

Table 6.3: Results of the case study on two spatio-temporal high-dimensional indices which are compared with the IBPC using PTAF.

(a) Achieved branch coverage (c_{ij}) of the test sets generated for the different indices (j) on the indices (i).

test set for (j)	executed on (i) RST-tree	STPA
RST-tree	0.512327	0.46413
STPA	0.490755	0.516304

(b) Relative coverage (cov_{ij}) of the test sets generated for the different indices (j) on the indices (i).

test set for (j)	executed on (i) RST-tree	STPA
RST-tree	1.0	0.898947
STPA	0.957895	1.0

(c) Achieved performance (p_{ij}) of the test sets generated for the different indices (j) on the indices (i) in s.

test set for (j)	executed on (i) RST-tree	STPA
RST-tree	0.167661	0.0951223
STPA	0.507944	0.378269

(d) Combined coverage and weighted performance with $w = 0$ (left) and $w = 30$ (right)

index	combined coverage	weighted performance [s]
RST-tree	1.0	0.337803
STPA	1.0	0.236696

index	combined coverage	weighted performance [s]
RST-tree	0.806471	1.00694
STPA	0.587045	1.35118

diverse data sets. Therefore, the conclusion that the STPA is the more efficient structure holds. In addition, the IBPC is able to reflect rather complex issues in the performance comparison of different structures.

Both, the configuration of the test generator and the structure of the competitors influence the outcome of the IBPC. For instance, the minimization of the generated test sets by only using the branch coverage and not the original fitness function for comparison, results in test sets with the same branch coverage but with a different behavior. For that case, the performance of the RST-tree is comparable to that of the STPA, as the missing of an increased number of loop iterations and method invocations influences the length of the test sets and especially the number of used elements. For a very small number of elements, the difference in the performance of the RST-tree and the STPA becomes very vague. If the set of competitors in this case study is extended by the sequential scan, the comparison of the RST-tree and the STPA remains the same but the sequential scan is identified as best performing competitor. This results from two facts: First, the sequential scan is covered by all test sets maximal as

it computes any element in the very same way. Secondly, for a small number of elements, the sequential scan works as efficient as the other indices as it does not use any further computations on the elements that are required to index the data.

Summing up, the case study of the IBPC using the spatio-temporal indices shows the following: First, the quality of the IBPC relies on the quality of the generated test sets. Secondly, the results of the classic workload generator can be approved by the IBPC. Thirdly, due to its comprehensive approach on performance testing, the IBPC unfolds performance issues on the STPA that are not reflected by the classic approach.

6.2.2 GNU Trove Library

This work focuses on efficient high-dimensional spatio-temporal indices and especially the evaluation and comparison of their performance. In order to show the capabilities of the IBPC, a rather simple set of structures is chosen in this case study. The GNU Trove library [23] provides a large set of abstract data types for the access on primitive types in Java. For instance, instead of using the `java.util.LinkedList` from the Java standard library and a wrapper class to store primitives, the Trove library implements a linked list for each primitive type. Avoiding the usage of autoboxing and -unboxing increases the performance of such structures significantly [23]. Here, three abstract data types, that store `byte` values are compared: a linked list, an array list and a hash set. For each of them a test set is created using the default settings described in Section 6.1.1, the ϵMOEA algorithm, no minimization, and $k = 1.25$.

The SUTs simply consists of the classes of the ADT themselves and no additional classes. The test interface for the generation of the test sets is defined to reflect the basic usage of the structures, the insertion, deletion and lookup of values: `add(byte)`, `delete(byte)` and `contains(byte)`. The performance of the test sets is measured using the performance measurement technique developed in this thesis (Section 5.4). The mean value of the performance measurements in 100 different (JVM) invocations is taken as the performance of a certain test set executed on one of the structures.

The tables in Table 6.4 show all values computed by the IBPC on the generated test sets. All test sets executed on the hash set have the same, maximum coverage. The test set generated of the array list has a lower coverage than the maximum when executed on the linked list. Both, the test set generated for the linked list and the test set generated for the hash set have a lower coverage than the maximum when executed on the array list (a + b). The performance of all three competitors is comparable for each test set as the standard deviations of all measurements overlap one another (c). For a higher weight of the coverage, the IBPC indicates that the hash set outperforms the linked list which outperforms the array list. That means that the hash set shows the overall best performance regarding the given interface. Only given `add(byte)`, `delete(byte)` and `contains(byte)`, one can easily agree on this outcome: For a higher number of elements, the array list needs to be rebuilt if a new, greater array has to be initialized. The execution of `contains(byte)` requires the array list and the linked list to sequentially lookup each element whereas the lookup on the hash set lies in $O(1)$.

Table 6.4: Results of the case study on three simple abstract data structures of the GNU Trove library [23] which are compared with the IBPC using PTAF.

(a) Achieved branch coverage (c_{ij}) of the test sets generated for the structures (j) on the structures (i).

test set for (j)	executed on (i) array list	linked list	hash set
array list	0.091716	0.0990099	0.0406977
linked list	0.0857988	0.10396	0.0406977
hash set	0.0857988	0.10396	0.0406977

(b) Relative coverage (cov_{ij}) of the test sets generated for the structures (j) on the structures (i).

test set for (j)	executed on (i) array list	linked list	hash set
array list	1.0	0.952381	1.0
linked list	0.935484	1.0	1.0
hash set	0.935484	1.0	1.0

(c) Achieved performance (p_{ij}) of the test sets generated for the structures (j) on the structures (i) in ms.

test set for (j)	executed on (i) array list	linked list	hash set
array list	0.829929	0.782302	0.812044
linked list	0.796448	0.808828	0.78213
hash set	0.505674	0.490927	0.518285

(d) Combined coverage and weighted performance with $w = 0$ (left) and $w = 10$ (right)

structure	combined coverage	weighted performance [ms]
array list	1.0	0.710683
linked list	1.0	0.694019
hash set	1.0	0.704153

structure	combined coverage	weighted performance [ms]
array list	0.641075	1.12225
linked list	0.849902	0.858014
hash set	1.0	0.704153

6.3 Conclusions

From the evaluation of the techniques developed in this thesis, most importantly the automated performance test generation and comparison, the following conclusion can be drawn:

- In terms of coverage, the Interface Based Test Generator (IBTG) generates test sets that are as good and better as the workloads generated by the classic workload generator.
- Equal to the existing test generators, the quality of the results of the IBTG depend on a proper configuration.

- The Interface Based Performance Comparison (IBPC) technique is able to automatically identify and compare a set of competitors with a common interface as expected. In the case of spatio-temporal indices, the IBPC unfolds performance issues that are not addressed by the classic workload generator.
- The results of the IBPC rely on the configuration of the test generator and on the definition of the test interface.
- The IBPC does not replace existing benchmarks but should be used in addition to existing benchmarks or for a set of competitors for which no proper benchmark exists.

Chapter 7

Summary and Future Work

This thesis introduces a new indexing technique for spatio-temporal high-dimensional data, the Spatio-Temporal Pyramid Adapter (STPA). In order to evaluate the new technique and compare its performance to the best competitor, the R^{ST}-tree [172], not only a classic workload generator (Chapter 3) is used, but a complete new system for the performance comparison of a set of competitors with a common interface is introduced. The new Interface Based Performance Comparison (IBPC, Chapter 4) technique uses two new metrics: the *combined coverage* to describe the comparability of the competitors and the *weighted performance* to measure the overall performance of the competitors in comparison to each other. The implementation of the IBPC, the Performance Test Automation Framework (PTAF , Chapter 5), describes a new adaptable system for the computation of control flow based coverage in Java, a new system for the generation of suitable performance tests in Java and a new system for the robust and reproducible performance measurement of Java programs. Here, performance is considered to be the computation time of a program on the CPU. The evaluation of the IBPC and PTAF (Chapter 6) shows that the automatically generated performance tests have a branch coverage as high and higher as the branch coverage of the classic workload generator. Most important, applying the IBPC on the comparison of the STPA and the R^{ST}-tree shows that overall the R^{ST}-tree works more stable considering different distributions of not only the spatial objects but especially the distribution of the *now*-relative values and their relation to the current value of *now*. Nonetheless, the STPA is more efficient for its intended use which is reflected by the classic workload generator. That shows the benefits of the IBPC: it unfolds all differences in a set of competitors automatically. It may be used to give an overview over the complete capabilities of a new structure in comparison to existing approaches. In addition, the IBPC generates a well fitting set of performance tests which may be used as a basis for the generation of specialized benchmarks.

All new techniques and implementations have a prototypal character. They suffice for the application in the desired domains but may not be suitable for a general use in the software industry. Nonetheless, using the outcome of this thesis, the STPA, IBPC and PTAF may easily be re-implemented for commercial use. The STPA may be used in the building information modeling (BIM) environment as an efficient backend for the storage of building models as well as in geographic information systems (GIS) for the storage of land and real estate registers or city models. The IBPC and PTAF can be used especially in the research of complex data structures for a most likely unbiased comparison of new techniques as well as for the comparison of programs in other domains. Besides, PTAF provides efficient implementations

for control based coverage computation, test generation and performance measurement in Java which may be used in other domains as well.

For the future research, the following issues are of most importance: The automatic adaption of the STPA to different distributions of data like in the P^+-tree [212] is only assumed and needs to be implemented and evaluated to prove the general efficiency of the STPA. The performance comparison by the IBPC using the combined coverage and weighted performance are a first shot for a generally fair performance comparison and both metrics may be replaced by other comparison methods without falsifying the outcome of this thesis. The efficiency of the coverage computation in this thesis suffices for the desired scope but may be enhanced significantly by using another type of probe than the method invocations used (Section 5.1). Like the already existing test generators, the quality of the test sets generated by the Interface Based Test Generator (IBTG) (Section 5.2) depends on its configuration and the used algorithms. A more exhaustive study like the study by Graf (2017) [91] for EVOSUITE [81] could lead to a more stable and generally efficient configuration. The new framework for a robust and reproducible performance measurement in Java (Section 5.4) depends on a preferably comprehensive setup of the parameters for a stable detection of the steady state of the performance measurements. As the parameters used in this thesis base on the evaluated index structures, before a use of the system for Java in general, the parameters should be refined using a larger evaluation.

The STPA [134] and the general idea of the IBPC and PTAF [133] already have been published and presented to a greater audience of experts in their respective domains. Besides, the generation of high covering test sets using an interface has been proven and published using GUI tests in [135]. This shows the general acknowledgement of the work presented in this thesis by the respective research communities.

Bibliography

[1] Business Applications Performance Corporation (BAPCo). URL https://bapco.com. Last Access: May 4th, 2018.

[2] Atlassian Clover. URL https://bitbucket.org/atlassian/clover. Last Access: May 4th, 2018.

[3] Coremark. URL http://www.eembc.org/coremark. Last Access: May 4th, 2018.

[4] Embedded Mircorprocessor benchmarks (EEMBC). URL http://www.eembc.org. Last Access: May 4th, 2018.

[5] Oracle Java HotSpot Engine. URL http://www.oracle.com/technetwork/Java/whitepaper-135217.html. Last Access: May 4th, 2018.

[6] IBM JIT compilation overview. URL https://www.ibm.com/support/knowledgecenter/SSYKE2_8.0.0/com.ibm.java.lnx.80.doc/diag/understanding/jit_optimize.html. Last Access: May 4th, 2018.

[7] Autotest for Java. URL https://sourceforge.net/projects/jautotest/. Last Access: May 4th, 2018.

[8] JCover, . URL https://mmsindia.com/products/. Last Access: May 4th, 2018.

[9] JaCoco Implementation Design, . URL http://www.jacoco.org/jacoco/trunk/doc/implementation.html. Last Access: May 4th, 2018.

[10] Java Spatial Index (JSI). URL https://github.com/aled/jsi. Last Access: May 4th, 2018.

[11] List of Java Virtual Machines, . URL https://en.wikipedia.org/w/index.php?title=List_of_Java_virtual_machines&oldid=794017935. Last Access: May 4th, 2018.

[12] Java Virtual Machine Profiler Interface, . URL http://download.oracle.com/otn_hosted_doc/jdeveloper/904preview/jdk14doc/docs/guide/jvmpi/jvmpi.html. Last Access: May 4th, 2018.

[13] Java Virtual Machine Tool Interface, . URL https://docs.oracle.com/javase/8/docs/technotes/guides/jvmti/index.html. Last Access: May 4th, 2018.

[14] KLEE website. URL http://klee.github.io. Last Access: May 4th, 2018.

[15] Open JDK. URL http://openjdk.Java.net/. Last Access: May 4th, 2018.

[16] Q-Up Data Generator. URL https://www.q-up-data.com/language/en/home-2/. Last Access: May 4th, 2018.

[17] SBST Contest. URL http://sbstcontest.dsic.upv.es. Last Access: May 4th, 2018.

[18] Storage Performance Council (SPC). URL http://www.storageperformance.org. Last Access: May 4th, 2018.

[19] Standard Performance Council (SPEC). URL http://www.spec.org. Last Access: May 4th, 2018.

[20] TestwellCTC++. URL http://www.verifysoft.com/en_ctcpp.html. Last Access: May 4th, 2018.

[21] TIOBE Index. URL https://www.tiobe.com/tiobe-index/. Last Access: May 4th, 2018.

[22] Transaction Processing Council (TPC). URL http://www.tpc.org. Last Access: May 4th, 2018.

[23] GNU Trove for Java. URL https://bitbucket.org/trove4j/trove. Last Access: May 4th, 2018.

[24] XXL library. URL https://github.com/umr-dbs/xxl. Last Access: May 4th, 2018.

[25] SBST Unit Testing Tool Competition 2013, 2013. URL http://sbstcontest.dsic.upv.es/?p=44. Last Access: May 4th, 2018.

[26] J. A, S. M, and R. R S. Indexing and Query Processing Techniques in Spatio-Temporal Data. *ICTACT Journal on Soft Computing*, 06(03):1198–1217, Apr. 2016.

[27] T. Abraham and J. F. Roddick. Survey of Spatio-Temporal Databases. *GeoInformatica*, 3(1):61–99, 1999.

[28] A. Aburas and A. Groce. A Method Dependence Relations Guided Genetic Algorithm. In F. Sarro and K. Deb, editors, *Search Based Software Engineering: Proceedings of the 8th International Symposium on Search Based Software Engineering*, pages 267–273. Springer International Publishing, Oct. 2016.

[29] A. Aleti, I. Moser, and L. Grunske. Analysing the fitness landscape of search-based software testing problems. *Automated Software Engineering*, 24(3):603–621, 2017.

[30] H. M. Alghmadi, M. D. Syer, W. Shang, and A. E. Hassan. An Automated Approach for Recommending When to Stop Performance Tests. In *Proceedings of the 2016 IEEE International Conference on Software Maintenance and Evolution*, pages 279–289. IEEE, 2016.

[31] J. F. Allen. Maintaining Knowledge About Temporal Intervals. *Communications of the ACM*, 26(11):832–843, Nov. 1983.

[32] L. Altenberg. The Evolution of Evolvability in Genetic Programming. In *Advances in genetic programming*, pages 47–74. Nov. 1994.

[33] A. Amighi, P. de Carvalho Gomes, D. Gurov, and M. Huisman. Provably correct control flow graphs from Java bytecode programs with exceptions. *International Journal on Software Tools for Technology Transfer*, 18(6):653–684, 2016.

[34] S. Anand, M. Naik, M. J. Harrold, and H. Yang. Automated concolic testing of smartphone apps. In *Proceedings of the ACM SIGSOFT 20th International Symposium*. ACM Press, 2012.

[35] L. Anselma, B. Stantic, P. Terenziani, and A. Sattar. Querying now-relative data. *Journal of Intelligent Information Systems*, 41(2):285–311, 2013.

[36] A. Arcuri. A Theoretical and Empirical Analysis of the Role of Test Sequence Length in Software Testing for Structural Coverage. *IEEE Transactions on Software Engineering*, 38(3):497–519, Apr. 2011.

[37] W. G. Aref and I. F. Ilyas. SP-GiST: An Extensible Database Index for Supporting Space Partitioning Trees. *Journal of Intelligent Information Systems*, 17(2/3):215–240, 2001.

[38] A. Avritzer, J. Kondek, D. Liu, and E. J. Weyuker. Software performance testing based on workload characterization. In *Proceedings of the 3rd international workshop on Software and performance*, pages 17–24. ACM, July 2002.

[39] L. Baresi, P. L. Lanzi, and M. Miraz. TestFul: An Evolutionary Test Approach for Java. In *Proceddings of the Third International Conference on Software Testing, Verification and Validation*, pages 185–194. IEEE, 2010.

[40] E. T. Barr, M. Harman, P. McMinn, M. Shahbaz, and S. Yoo. The Oracle Problem in Software Testing: A Survey. *IEEE Transactions on Software Engineering*, 41(5): 507–525, May 2015.

[41] S. Bauersfeld, T. E. J. Vos, and K. Lakhotia. Unit Testing Tool Competitions - Lessons Learned. In *Future Internet Testing: Proceedings of the First International Workshop on Future Internet Testing*, pages 75–94. Springer International Publishing, 2013.

[42] S. Bauersfeld, T. E. J. Vos, K. Lakhotia, S. Poulding, and N. Condori. Unit Testing Tool Competition. In *Proceedings of the IEEE 6th International Conference On Software Testing, Verification and Validation Workshops*, pages 414–420. IEEE, July 2013.

[43] P. D. R. Bayer and D. E. M. McCreight. Organization and Maintenance of Large Ordered Indexes. *Acta Informatica*, 1(3):173–189, 1972.

[44] N. Beckmann, H.-P. Kriegel, R. Schneider, and B. Seeger. The R*-tree: An Efficient and Robust Access Method for Points and Rectangles. In *Proceedings of the 1990 ACM SIGMOD international conference on Management of data*, pages 322–331. ACM, 1990.

[45] M. Behrisch, L. Bieker, J. Erdmann, and D. Krajzwicz. Sumo (simulation of urban mobility). In *Proceedings of the 3rd International Conference on Advances in System Simulation*, pages 55–60, Oct. 2011.

[46] S. Berchtold, D. A. Keim, and H.-P. Kriegel. The X-Tree: An Index Structure for High-Dimensional Data. In *Proceedings of the 22nd International Conference on Very Large Databases*, pages 28–39. Morgan Kaufmann, 1996.

[47] S. Berchtold, C. Böhm, and H.-P. Kriegel. The Pyramid-technique: Towards Breaking the Curse of Dimensionality. In *Proceedings of the 1998 ACM SIGMOD international conference on Management of data*, pages 142–153. ACM, 1998.

[48] A. Bergel, J. P. S. Alcocer, and A. Bergel. Tracking down performance variation against source code evolution. In *Proceedings of the 2013 Symposium on memory management*, pages 129–139. ACM, May 2016.

[49] F. Biljecki, H. Ledoux, and J. E. Stoter. An improved LOD specification for 3D building models. *Computers, Environment and Urban Systems*, 59:25–37, 2016.

[50] A. Borrmann, T. H. Kolbe, A. Donaubauer, H. Steuer, J. R. Jubierre, and M. Flurl. Multi-Scale Geometric-Semantic Modeling of Shield Tunnels for GIS and BIM Applications. *Computer-Aided Civil and Infrastructure Engineering*, 30(4):263–281, 2015.

[51] B. Boyer. Robust Java benchmarking, June 2008. URL `https://www.ibm.com/developerworks/java/library/j-benchmark1/index.html`. Last Access: May 4th, 2018.

[52] M. Breunig, A. Borrmann, E. Rank, S. Hinz, T. Kolbe, M. Schilcher, R. P. Mundani, J. R. Jubierre, M. Flurl, A. Thomsen, A. Donaubauer, Y. Ji, S. Urban, S. Laun, S. Vilgertshofer, B. Willenborg, M. Menninghaus, H. Steuer, S. Wursthorn, J. Leitloff, M. Al-Doori, and N. Mazroobsemnani. Collaborative Multi-Scale 3D City and Infrastructure Modeling and Simulation. *ISPRS - International Archives of the Photogrammetry, Remote Sensing and Spatial Information Sciences*, XLII-4/W4:341–352, 2017.

[53] T. Brinkhoff. A Framework for Generating Network-Based Moving Objects. *GeoInformatica*, 6(2):153–180, June 2002.

[54] S. Brown, A. Mitchell, and J. F. Power. A Coverage Analysis of Java Benchmark Suites. In *Proceedings of the 2013 International Conference on Software Engineering*, 2005.

[55] L. Bulej, T. Bureš, V. Horký, J. Kotrč, L. Marek, T. Trojánek, and P. Tůma. Unit testing performance with Stochastic Performance Logic. *Automated Software Engineering*, 1(24): 139–187, Jan. 2016.

[56] J. Burnim, S. Juvekar, and K. Sen. WISE: Automated test generation for worst-case complexity. In *Proceedings of the 31st International Conference on Software Engineering*, pages 463–473. IEEE, 2009.

[57] C. Cadar, D. Dunbar, and D. R. Engler. KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs. In *Proceedings of the 8th USENIX conference on Operating systems*, pages 209–224, 2008.

[58] C. Cadar, V. Ganesh, P. M. Pawlowski, D. L. Dill, and D. R. Engler. EXE: Automatically Generating Inputs of Death. *ACM Transactions on Information and System Security*, 12(2):10–38, Dec. 2008.

[59] J. Campos, Y. Ge, G. Fraser, M. Eler, and A. Arcuri. An Empirical Evaluation of Evolutionary Algorithms for Test Suite Generation. In T. Menzies and J. Petke, editors, *Search Based Software Engineering: Proceedings of the 9th International Symposium on Search Based Software Engineering*, pages 33–48. Springer International Publishing, 2017.

[60] S. Carino. *Dynamically Testing Graphical User Interfaces*. PhD thesis, Electronic Thesis and Dissertation Repository, 2016.

[61] S. Cass. The 2017 Top Programming Languages. URL `http://spectrum.ieee.org/computing/software/the-2017-top-programming-languages`. Last Access: May 4th, 2018.

[62] Y. M. Chee, C. J. Colbourn, D. Horsley, and J. Zhou. Sequence Covering Arrays. *SIAM Journal on Discrete Mathematics*, 27(4):1844–1861, Oct. 2013.

[63] M. Chen, X. Qiu, W. Xu, L. Wang, J. Zhao, and X. Li. UML Activity Diagram-Based Automatic Test Case Generation For Java Programs. *The Computer Journal*, 52(5): 545–556, Aug. 2009.

[64] S. Chen, C. S. Jensen, and D. Lin. A Benchmark for Evaluating Moving Object Indexes. *Proceedings of the VLDB Endowment*, 1(2):1574–1585, Aug. 2008.

[65] S. Christou. Cobertura. URL `http://cobertura.github.io/cobertura/`. Last Access: May 4th, 2018.

[66] J. Clifford, C. Dyreson, T. a. s. Isakowitz, C. S. Jensen, and R. T. Snodgrass. On the Semantics of Now in Databases. *ACM Transactions on Database Systems*, 22(2): 171–214, June 1997.

[67] D. Comer. The Ubiquitous B-Tree. *ACM Computing Surveys*, 11(2):121–137, June 1979.

[68] D. W. Corne, N. R. Jerram, J. D. Knowles, M. J. Oates, and M. J. PESA-II: Region-based Selection in Evolutionary Multiobjective Optimization. In *Proceedings of the Genetic and Evolutionary Computation Conference*, pages 283–290, 2001.

[69] M. Couck. Serenity. URL `https://wiki.jenkins.io/display/JENKINS/Serenity+Plugin`. Last Access: May 4th, 2018.

[70] C. Csallner and Y. Smaragdakis. JCrasher: an automatic robustness tester for Java. *Software: Practice and Experience*, 34(11):1025–1050, Sept. 2004.

[71] H. Dan, M. Harman, J. Krinke, L. Li, and A. Marginean. Pidgin Crasher: Searching for Minimised Crashing GUI Event Sequences. In C. Le Goues and S. Yoo, editors, *Search Based Software Engineering: Proceedings of the 6th International Symposium on Search Based Software Engineering*, pages 253–258, 2014.

[72] K. Deb. An efficient constraint handling method for genetic algorithms. *Computer Methods in Applied Mechanics and Engineering*, 186(2-4):311–338, June 2000.

[73] K. Deb and R. B. Agrawal. Simulated binary crossover for continuous search space. *Complex Systems*, 9:1–34, 1994.

[74] K. Deb and M. Goyal. A combined genetic adaptive search (GeneAS) for engineering design. *Computer Science an Informatics*, 26(4):30–45, 1996.

[75] K. Deb, S. Agrawal, A. Pratap, and T. Meyarivan. A fast and elitist multiobjective genetic algorithm: NSGA-II. *IEEE Transactions Evolutionary Computation*, 6(2):182–197, July 2002.

[76] K. Deb, M. Mohan, and S. Mishra. Towards a Quick Computation of Well-Spread Pareto-Optimal Solutions. In *Evolutionary Multi-Criterion Optimization Proceedings of the Second International Conference on Evolutionary Multi-Criterion Optimization*, pages 222–236. Springer, Berlin, Heidelberg, Apr. 2003.

[77] C. Düntgen, T. Behr, and R. H. Güting. BerlinMOD - a benchmark for moving object databases. *The VLDB Journal*, 18(6):1335–1368, 2009.

[78] M. Egenhofer. A Formal Definition of Binary Topological Relationships. In *Foundations of Data Organization and Algorithms: Proceedings of the 3rd International Conference on Foundations of Data Organizations and Algorithms*, pages 457–472. Springer Berlin Heidelberg, 1989.

[79] M. J. Egenhofer, K. C. Clarke, S. Gao, T. Quesnot, W. R. Franklin, M. Yuan, and D. Coleman. Contributions of GIScience over the Past Twenty Years. In *Advancing Geographic Information Science: The Past and Next Twenty Years*, pages 9–34. Feb. 2016.

[80] R. A. Finkel and J. L. Bentley. Quad Trees - A Data Structure for Retrieval on Composite Keys. *Acta Informatica*, 4(1):1–9, 1974.

[81] G. Fraser and A. Arcuri. It is Not the Length That Matters, It is How You Control It. In *Proceedings of the Fourth International Conference on Software Testing, Verification and Validation*, pages 150–159. IEEE, 2011.

[82] G. Fraser and A. Arcuri. Whole Test Suite Generation. *IEEE Transactions on Software Engineering*, 39(2):276–291, Jan. 2013.

[83] G. Fraser and A. Arcuri. EvoSuite at the Second Unit Testing Tool Competition. In *Proceedings of International Workshop on Future Internet Testing*, pages 1–6, Dec. 2013.

[84] A. Freitas and R. Vieira. An Ontology for Guiding Performance Testing. In *Proceedings of the IEEE/WIC/ACM International Conferences on Web Intelligence (WI) and Intelligent Agent Technologies (IAT)*, pages 400–407. IEEE Computer Society, Aug. 2014.

[85] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns.* Elements of Reusable Object-Oriented Software. Pearson Education, 1994.

[86] S. R. Ganov, C. Killmar, S. Khurshid, and D. E. Perry. Test generation for graphical user interfaces based on symbolic execution. In *Proceedings of the 3rd international workshop on Automation of software test*, pages 33–40. ACM, May 2008.

[87] A. Georges, D. Buytaert, L. Eeckhout, A. Georges, D. Buytaert, and L. Eeckhout. Statistically rigorous java performance evaluation. In *Proceedings of the 22nd annual*

ACM SIGPLAN conference on Object-oriented programming systems and applications, pages 57–76. ACM, Oct. 2007.

[88] P. Godefroid, N. Klarlund, and K. Sen. DART: directed automated random testing. In *Proceedings of the ACM SIGPLAN conference on Programming languages design and implementation*, pages 213–223. ACM, June 2005.

[89] A. Gotlieb. Euclide: A Constraint-Based Testing Framework for Critical C Programs. In *Proceedings of the 2009 International Conference on Software Testing Verification and Validation*, pages 151–160. IEEE, 2009.

[90] G. Graefe. Modern B-Tree Techniques. *Foundations and Trends in Databases*, 3(4): 203–402, 2011.

[91] B. Graf. Korrelation zwischen Testabdeckung, Software-Maßen und der Evolvierbarkeit in der automatisierten Testfallerzeugung. Master's thesis, Institute for Computer Science, University of Osnabrueck, 2017.

[92] M. Grechanik, C. Fu, and Q. Xie. Automatically finding performance problems with feedback-directed learning software testing. In *Proceedings of the 25th International Conference on Software Engineering*, pages 156–166. IEEE Press, June 2012.

[93] F. Gross, G. Fraser, and A. Zeller. EXSYST: search-based GUI testing. In *Proceedings of 34th International Conference on Software Engineering*, pages 1423–1426. IEEE Press, June 2012.

[94] F. Gross, G. Fraser, and A. Zeller. Search-based system testing: high coverage, no false alarms. In *Proceedings of the 2012 International Symposium on Software Testing and Analysis*, pages 67–77. ACM, July 2012.

[95] A. Guttman. R-trees: A Dynamic Index Structure for Spatial Searching. In *Proceedings of the 1984 ACM SIGMOD international conference on Management of data*, pages 47–57. ACM, 1984.

[96] D. Hadka. MOEA framework. URL `http://moeaframework.org/index.html`. Last Access: May 4th, 2018.

[97] Z. He, M.-J. Kraak, O. Huisman, X. Ma, and J. Xiao. Parallel indexing technique for spatio-temporal data. *International Journal of Photogrammetry and Remote Sensing*, 78(0):116–128, Apr. 2013.

[98] M. S. Hecht. *Flow Analysis of Computer Programs.* Elsevier North-Holland Inc., Jan. 1977.

[99] J. M. Hellerstein, J. F. Naughton, and A. Pfeffer. Generalized Search Trees for Database Systems. In *Proceedings of the 21st International Conference on Very Large Data Bases*, pages 562–573, 1995.

[100] D. W. Hoffmann. *Software-Qualität.* eXamen.press. Springer Berlin Heidelberg, 2 edition, 2013.

[101] M. R. Hoffmann, B. Janiczak, and E. Mandrikov. EclEmma. URL http://www.eclemma.org/index.html. Last Access: May 4th, 2018.

[102] J. Holland. Adaptation in artificial and natural systems: An introcutory analysis with applications to biology, control, and artificial intelligence. The MIT Press, Cambridge, Masschusetts, 1992.

[103] V. Horký, P. Libič, L. Marek, A. Steinhauser, and P. Tůma. Utilizing Performance Unit Tests To Increase Performance Awareness. In *Proceedings of the 6th ACM/SPEC International Conference on Performance Engineering*, pages 289–300. ACM, Jan. 2015.

[104] K. Huppler. The Art of Building a Good Benchmark. In R. Nambiar and M. Poess, editors, *Performance Evaluation and Benchmarking: Proceedings of the First TPC Technology Conference*, pages 18–30. Springer Berlin Heidelberg, 2009.

[105] H. V. Jagadish, B. C. Ooi, K.-L. Tan, C. Yu, and R. Zhang. iDistance: An Adaptive B+-Tree Based Indexing Method for Nearest Neighbor Search. *ACM Transactions on Database Systems*, 30(2):364–397, June 2005.

[106] R. Jain. *The art of computer systems performance analysis*. Techniques for experimental design, measurement, simulation, and modeling. John Wiley & Sons, Inc., 1991.

[107] C. S. Jensen, D. Lin, and B. C. Ooi. Query and Update Efficient B+-Tree Based Indexing of Moving Objects. In *Proceedings of the 13th international conference on Very Large Data Bases*, pages 768–779. VLDB Endowment, 2004.

[108] C. S. Jensen, D. Tiešytė, and N. Tradišauskas. The COST Benchmark—Comparison and Evaluation of Spatio-temporal Indexes. In *Database Systems for Advanced Applications: Proceedings of the 11th International Conference on Database Systems for Advanced Applications*, pages 125–140. Springer Berlin Heidelberg, 2006.

[109] C. S. Jensen, M. R. Prasad, and A. Møller. Automated testing with targeted event sequence generation. In *Proceedings of the 2013 International Symposium on Software Testing and Analysis*, pages 67–77. ACM, July 2013.

[110] T. Kalibera and R. Jones. Rigorous benchmarking in reasonable time. In *Proceedings of the 2013 Symposium on memory management*, pages 63–74. ACM, Dec. 2013.

[111] M. Kempka. Coverlipse. URL http://coverlipse.sourceforge.net/index.php. Last Access: May 4th, 2018.

[112] T. H. Kolbe. Representing and Exchanging 3D City Models with CityGML. In *3D Geoinformation Sciences*, pages 15–31. Springer Berlin Heidelberg, 2009.

[113] M. Köppen and K. Yoshida. Substitute Distance Assignments in NSGA-II for Handling Many-objective Optimization Problems. In S. Obayashi, K. Deb, C. Poloni, T. Hiroyasu, and T. Murata, editors, *Evolutionary Multi-Criterion Optimization: Proceedings of the 4th International Conference on Evolutionary Multi-Criterion Optimization*, pages 727–741. Springer Berlin Heidelberg, 2007.

[114] V. Köppen, M. Schäler, and R. Schröter. Toward Variability Management to Tailor High Dimensional Index Implementations. In *Proceedings of the IEEE Eighth International Conference on Research Challenges in Information Science*, pages 1–6. IEEE, May 2014.

[115] B. Korel. Automated software test data generation. *IEEE Transactions on Software Engineering*, 16(8):870–879, 1990.

[116] H.-P. Kriegel, M. Pötke, and T. Seidl. Interval Sequences: An Object-Relational Approach to Manage Spatial Data. In C. S. Jensen, M. Schneider, B. Seeger, and V. J. Tsotras, editors, *Advances in Spatial and Temporal Databases: Proceedings of the 7th International Symposium on Spatial and Temporal Databases*, pages 481–501. Springer Berlin Heidelberg, 2001.

[117] M. Laumanns, L. Thiele, K. Deb, and E. Zitzler. Combining Convergence and Diversity in Evolutionary Multiobjective Optimization. *Evolutionary Computation*, 10(3):263–282, Mar. 2002.

[118] A. Leitner, I. Ciupa, O. Manuel, B. Meyer, and A. Fiva. Contract driven development = test driven development - writing test cases. In *Proceedings of the the 6th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering*, pages 425–434. ACM Press, 2007.

[119] B. Li, J. Li, K. Tang, and X. Yao. Many-Objective Evolutionary Algorithms. *ACM Computing Surveys*, 48(1):1–35, Sept. 2015.

[120] H. Li and Q. Zhang. Multiobjective Optimization Problems With Complicated Pareto Sets, MOEA/D and NSGA-II. *IEEE Transactions on Evolutionary Computation*, 13(2): 284–302, Apr. 2009.

[121] T. Liebich, Y. Adachi, J. Forester, J. Hyvarinen, S. Richter, T. Chipman, M. Weise, and J. Wix. Industry Foundation Classes Release 4 (IFC4), 2013. URL `http://www.buildingsmart-tech.org/ifc/IFC4/final/html/index.htm`. Last Access: May 4th, 2018.

[122] P. Liggesmeyer. Software-Qualität: Testen, Analysieren und Verifizieren von Software. Heidelberg, Berlin, 2002.

[123] K.-I. Lin, H. V. Jagadish, and C. Faloutsos. The TV-tree: An Index Structure for High-Dimensional data. *The VLDB Journal*, 3(4):517–542, 1994.

[124] T. Lindholm, F. Yelin, G. Bracha, and A. Buckley. The Java Virtual Machien Specification, Feb. 2015. URL `https://docs.oracle.com/javase/specs/jvms/se8/html/index.html`. Last Access: May 4th, 2018.

[125] K. Looney. JCov. URL `https://wiki.openjdk.java.net/display/CodeTools/jcov`. Last Access: May 4th, 2018.

[126] L. Ma, C. Artho, C. Zhang, H. Sato, J. Gmeiner, and R. Ramler. GRT - Program-Analysis-Guided Random Testing (T). In *Proceedings of the 30th IEEE/ACM International Conference on Automated Software Engineering*, pages 212–223. IEEE, 2015.

[127] D. Marinov, C. Boyapati, and S. Khurshid. Korat: automated testing based on Java predicates. In *Proceedings of the 2002 ACM SIGSOFT international symposium on Software testing and analysis*, pages 123–133. ACM, July 2002.

[128] P. McMinn. Search-based software test data generation: a survey. *Software Testing, Verification and Reliability*, 14(2):105–156, June 2004.

[129] P. McMinn. Search-Based Software Testing: Past, Present and Future. In *Proceedings of the Fourth International Conference on Software Testing, Verification and Validation Workshops*, pages 153–163. IEEE, 2011.

[130] D. Meagher. Geometric modeling using octree encoding. *Computer Graphics and Image Processing*, 19(2):129–147, 1982.

[131] A. M. Memon. *A comprehensive framework for testing graphical user interfaces*. PhD thesis, 2001.

[132] A. M. Memon. An event-flow model of GUI-based applications for testing. *Software Testing, Verification and Reliability*, 17(3):137–157, 2007.

[133] M. Menninghaus and E. Pulvermüller. Towards Using Code Coverage Metrics for Performance Comparison on the Implementation Level. In *Proceedings of the 7th ACM/SPEC International Conference on Performance Engineering*, pages 101–104. ACM, Mar. 2016.

[134] M. Menninghaus, M. Breunig, and E. Pulvermüller. High-Dimensional Spatio-Temporal Indexing. *Open Journal of Databases (OJDB)*, 3(1):1–20, 2016.

[135] M. Menninghaus, F. Wilke, J.-P. Schleutker, and E. Pulvermüller. Search based GUI Test Generation in Java. In *Proceedings of the 12th International Conference on Evaluation of Novel Approaches to Software Engineering*, pages 179–186. SCITEPRESS - Science and Technology Publications, 2017.

[136] N. Meyering. Analysis and Implementation of a Path Coverage Metric in Java. Master's thesis, Institute for Computer Science, University of Osnabrueck, 2016.

[137] A. Milicevic, S. Misailovic, D. Marinov, and S. Khurshid. Korat: A Tool for Generating Structurally Complex Test Inputs. In *Proceedings of the 29th International Conference on Software Engineering*, pages 771–774. IEEE Computer Society, May 2007.

[138] M. F. Mokbel, T. M. Ghanem, and W. G. Aref. Spatio-Temporal Access Methods. *IEEE Data Engineering Bulletin*, 26(2):40–49, 2003.

[139] D. Moten. Moten's R-Tree. URL `https://github.com/davidmoten/rtree`. Last Access: May 4th, 2018.

[140] M. A. Nascimento and J. R. O. Silva. Towards Historical R-trees. In *Proceedings of the 1998 ACM symposium on Applied Computing*, pages 235–240. ACM, 1998.

[141] M. A. Nascimento, J. R. O. Silva, and Y. Theodoridis. Evaluation of Access Structures for Discretely Moving Points. In *Proceedings of the International Workshop on Spatio-Temporal Database Management*, pages 171–189. Springer Berlin Heidelberg, 1999.

[142] B. N. Nguyen, B. Robbins, I. Banerjee, and A. Memon. GUITAR: an innovative tool for automated testing of GUI-driven software. *Automated Software Engineering*, 21(1): 65–105, 2014.

[143] L. V. Nguyen-Dinh, W. G. Aref, and M. Mokbel. Spatio-Temporal Access Methods: Part 2 (2003-2010). *IEEE Data Engineering Bulletin*, 33(2):46–55, 2010.

[144] J. Nievergelt, H. Hinterberger, and K. C. Sevcik. The Grid File: An Adaptable, Symmetric Multikey File Structure. *ACM Transactions on Database Systems*, 9(1):38–71, Mar. 1984.

[145] C. Oriat. Jartege: a tool for random generation of unit tests for java classes. In *Proceedings of the First international conference on Quality of Software Architectures and Software Quality, and Proceedings of the Second International conference on Software Quality*, pages 242–256, 2005.

[146] C. Pacheco and M. D. Ernst. Randoop: feedback-directed random testing for Java. In *Companion to the 22nd ACM SIGPLAN conference on Object-oriented programming systems and applications*, pages 815–816. ACM, Oct. 2007.

[147] C. Pacheco, S. K. Lahiri, M. D. Ernst, and T. Ball. Feedback-Directed Random Test Generation. In *Proceedings of the 29th International Conference on Software Engineering*, pages 75–84. IEEE, 2007.

[148] A. Panichella, F. M. Kifetew, and P. Tonella. Reformulating Branch Coverage as a Many-Objective Optimization Problem. In *Proceedings of the First International Conference on Software Testing, Verification and Validation*, pages 1–10. IEEE, 2015.

[149] A. Panichella, F. Kifetew, and P. Tonella. Automated Test Case Generation as a Many-Objective Optimisation Problem with Dynamic Selection of the Targets. *IEEE Transactions on Software Engineering*, 44(2):122–158, Feb. 2017.

[150] C. S. Păsăreanu and W. Visser. A survey of new trends in symbolic execution for software testing and analysis. *International Journal on Software Tools for Technology Transfer*, 11(4):339–353, 2009.

[151] C. S. Păsăreanu, S. Khurshid, and W. Visser. Test input generation with java PathFinder. In *Proceedings of the 2002 ACM SIGSOFT international symposium on Software testing and analysis*, pages 97–107. ACM, July 2004.

[152] C. S. Păsăreanu, P. C. Mehlitz, D. H. Bushnell, K. Gundy-Burlet, M. Lowry, S. Person, and M. Pape. Combining unit-level symbolic execution and system-level concrete execution for testing nasa software. In *Proceedings of the 2008 international symposium on Software testing and analysis*, pages 15–26. ACM, July 2008.

[153] C. S. Păsăreanu, W. Visser, D. Bushnell, J. Geldenhuys, P. Mehlitz, and N. Rungta. Symbolic PathFinder: integrating symbolic execution with model checking for Java bytecode analysis. *Automated Software Engineering*, 20(3):391–425, Feb. 2013.

[154] J. M. Patel. Rethinking Benchmarking for Data. In R. Nambiar and M. Poess, editors, *Performance Evaluation and Benchmarking: Traditional to Big Data to Internet of*

Things: Proceedings of the 7th Technology Conference on Performance Evaluation And Benchmarking, pages 130–134. Springer, Cham, Aug. 2015.

[155] N. Pelekis, B. Theodoulidis, I. Kopanakis, and Y. Theodoridis. Literature review of spatio-temporal database models. *The Knowledge Engineering Review*, 19(03):235–274, Sept. 2004.

[156] D. J. Peuquet. A Conceptual Framework and Comparison of Spatial Data Models. *Cartographica: The International Journal for Geographic Information and Geovisualization*, 21(4):66–113, Oct. 1984.

[157] E. Pitzer and M. Affenzeller. A Comprehensive Survey on Fitness Landscape Analysis. In *Recent Advances in Intelligent Engineering Systems. Studies in Computational Intelligence*, pages 161–191. Springer Berlin Heidelberg, 2012.

[158] I. S. W. B. Prasetya. T3, a Combinator-Based Random Testing Tool for Java - Benchmarking. In T. Vos, K. Lakhotia, and S. Bauersfeld, editors, *Future Internet Testing: Proceedings of the First International Workshop on Future Internet Testing*, pages 101–110. Springer International Publishing, 2013.

[159] I. S. W. B. Prasetya. T3i - a tool for generating and querying test suites for Java. In *Proceedings of the 10th Joint Meeting on Foundations of Software Engineering*, pages 950–953. ACM Press, 2015.

[160] I. S. W. B. Prasetya, T. E. J. Vos, and A. I. Baars. Trace-based Reflexive Testing of OO Programs with T2. In *Proceedings of the First International Conference on Software Testing, Verification and Validation*, pages 151–160. IEEE, 2008.

[161] D. G. Reichelt and S. Kühne. Empirical Analysis of Performance Problems on Code Level. In *Proceedings of the 7th ACM/SPEC International Conference on Performance Engineering*, pages 117–120. ACM, Mar. 2016.

[162] J. T. Robinson. The K-D-B-Tree: A Search Structure for Large Multidimensional Dynamic Indexes. In *Proceedings of the 1981 ACM SIGMOD international conference on Management of data*, pages 10–18. ACM, 1981.

[163] M. Rodriguez-Cancio, B. Combemale, and B. Baudry. Automatic microbenchmark generation to prevent dead code elimination and constant folding. *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering*, pages 132–143, Aug. 2016.

[164] J. M. Rojas, G. Fraser, and A. Arcuri. Seeding strategies in search-based unit test generation. *Software Testing, Verification and Reliability*, 26(5):366–401, 2016.

[165] J. M. Rojas, M. Vivanti, A. Arcuri, and G. Fraser. A detailed investigation of the effectiveness of whole test suite generation. *Empirical Software Engineering*, 22(2):852–893, Mar. 2016.

[166] U. Rueda, T. E. J. Vos, and I. S. W. B. Prasetya. Unit Testing Tool Competition - Round Three. In *Proceedings of the IEEE/ACM 8th International Workshop on Search-Based Software Testing*, pages 19–24. IEEE, 2015.

[167] U. Rueda, R. Just, J. P. Galeotti, and T. E. J. Vos. Unit testing tool competition - round four. In *Proceedings of the IEEE/ACM 9th International Workshop on Search-Based Software Testing*, pages 19–28. ACM Press, 2016.

[168] S. J. Russell and P. Norvig. *Artificial intelligence: a modern approach, 3rd Edition*. Pearson, 2010.

[169] J.-M. Saglio and J. Moreira. Oporto: A Realistic Scenario Generator for Moving Objects. *GeoInformatica*, 5(1):71–93, Mar. 2001.

[170] A. Sakti, G. Pesant, and Y.-G. Guéhéneuc. Instance Generator and Problem Representation to Improve Object Oriented Code Coverage. *IEEE Transactions on Software Engineering*, 41(3):294–313, 2015.

[171] Y. D. Salman and N. L. Hashim. Automated Test Case Generation from UML State Chart Diagram: A Survey. *Advanced Computer and Communication Engineering Technology*, 362:1325, Dec. 2015.

[172] S. Saltenis and C. S. Jensen. R-tree Based Indexing of General Spatio-Temporal Data. Technical report, 1999.

[173] S. Saltenis and C. S. Jensen. Indexing of moving objects for location-based services. *Proceedings of the 18th International Conference on Data Engineering*, pages 463–472, 2002.

[174] S. Saltenis, C. S. Jensen, S. T. Leutenegger, and M. A. Lopez. Indexing the Positions of Continuously Moving Objects. In *Proceedings of the 2000 ACM SIGMOD international conference on Management of data*, pages 24–342. ACM, 2000.

[175] M. Schäler, A. Grebhahn, R. Schröter, S. Schulze, V. Köppen, and G. Saake. QuEval: Beyond high-dimensional indexing à la carte. *Proceedings of the VLDB Endowment*, 6 (14):1654–1665, 2013.

[176] R. Schmidberger. CodeCover. URL `http://codecover.org`. Last Access: May 4th, 2018.

[177] K. Sen and G. Agha. CUTE and jCUTE: Concolic Unit Testing and Explicit Path Model-Checking Tools. In T. Ball and R. B. Jones, editors, *Computer Aided Verification: Proceedings of the 2016 International Conference on Computer Aided Verification*, pages 419–423. Springer Berlin Heidelberg, Aug. 2006.

[178] K. Sen, D. Marinov, G. Agha, and G. Agha. CUTE: a concolic unit testing engine for C. In *Proceedings of the 10th European software engineering conference held jointly with the 13th ACM SIGSOFT international symposium on Foundations of software engineering*, pages 263–272. ACM, Sept. 2005.

[179] C. E. Shannon and W. Weaver. *The Mathematical Theory of Information*. Urbana Ilinois: University of Ilinois Press, 1949.

[180] R. Sharma, M. Gligoric, V. Jagannath, and D. Marinov. A Comparison of Constraint-Based and Sequence-Based Generation of Complex Input Data Structures. In *Proceedings*

of the Third International Conference on Software Testing, Verification, and Validation Workshops, pages 337–342. IEEE, Apr. 2010.

[181] D. Shen, Q. Luo, D. Poshyvanyk, and M. Grechanik. Automating performance bottleneck detection using search-based application profiling. In *Proceedings of the 2015 International Symposium on Software Testing and Analysis*, pages 270–281. ACM, July 2015.

[182] S. E. Sim, S. Easterbrook, and R. C. Holt. Using benchmarking to advance research: a challenge to software engineering. In *Proceedings of the 25th International Conference on Software Engineering*, pages 74–83. IEEE Computer Society, May 2003.

[183] C. U. Smith. Software Performance Engineering Then and Now: A Position Paper. In *Proceedings of the 2015 Workshop on Challenges in Performance Methods for Software Development*, pages 1–3. ACM, Jan. 2015.

[184] C. U. Smith and L. G. Williams. Software performance antipatterns. In *Proceedings of the 2nd International Workshop on Software and Performance*, 2000.

[185] R. T. Snodgrass and I. Ahn. A Taxonomy of Time in Databases. In *Proceedings of the 1985 ACM SIGMOD international conference on Management of data*, pages 236–246. ACM Press, 1985.

[186] B. Stantic, A. Sattar, and P. Terenziani. The POINT approach to represent now in bitemporal databases. *Journal of Intelligent Information Systems*, 32(3):297–323, 2009.

[187] B. Stantic, R. Topor, J. Terry, and A. Sattar. Advanced Indexing Technique for Temporal Data. *Computer Science and Information Systems*, 7(4):679–703, 2010.

[188] Y. Theodoridis, J. R. O. Silva, and M. A. Nascimento. On the Generation of Spatiotemporal Datasets. In *Proceedings of the 6th International Symposium on Advances in Spatial Databases*, pages 147–164. Springer-Verlag, 1999.

[189] P. Tonella. Evolutionary testing of classes. *ACM SIGSOFT Software Engineering Notes*, 29(4):119–128, July 2004.

[190] K. Torp, C. S. Jensen, and M. H. Böhlen. Layered Temporal DBMS - Concepts and Techniques. In *Proceedings of the Fifth International Conference on Database Systems for Advanced Applications*, pages 371–380, 1997.

[191] N. Tracey, J. Clark, K. Mander, and J. McDermid. An Automated Framework for Structural Test-Data Generation. In *Proceedings of the 13th IEEE international conference on Automated software engineering*, pages 285–289, Sept. 1998.

[192] T. Tzouramanis, M. Vassilakopoulos, and Y. Manolopoulos. Overlapping Linear Quadtrees: a Spatio-temporal Access Method. In *Proceedings of the 6th ACM international symposium on Advances in geographic information systems*, pages 1–7. ACM, 1998.

[193] T. Tzouramanis, M. Vassilakopoulos, and Y. Manolopoulos. On the Generation of Time-Evolving Regional Data. *GeoInformatica*, 6(3):207–231, Sept. 2002.

[194] P. Van Oosterom, W. Quak, and T. Tjissen. Testing current DBMS products with real spatial data. In *Proceedings of 23rd Urban Data Management Symposium*, pages 1–4, 2002.

[195] C. Vandevelde. Android-R-Tree. URL `https://github.com/cnvandev/Android-R-Tree`. Last Access: May 4th, 2018.

[196] C. Vatterodt. An Eclipse Plug-In for Calculating Condition Coverage in Java. Master's thesis, Institute for Computer Science, University of Osnabrueck, 2016.

[197] M. Vieira, J. Leduc, B. Hasling, R. Subramanyan, and J. Kazmeier. Automation of GUI testing using a model-driven approach. In *Proceedings of the 2006 international workshop on Automation of software test*, pages 9–14. ACM, May 2006.

[198] J. von Kistowski, J. A. Arnold, K. Huppler, K.-D. Lange, J. L. Henning, and P. Cao. How to Build a Benchmark. In *Proceedings of the 6th ACM/SPEC International Conference on Performance Engineering*, pages 333–336. ACM Press, 2015.

[199] C. von Lücken, B. Barán, and C. A. Brizuela. A survey on multi-objective evolutionary algorithms for many-objective problems. *Computational Optimization and Applications*, 58(3):707–756, 2014.

[200] J. Wang, J. Lu, Z. Fang, T. Ge, and C. Chen. PL-Tree: An Efficient Indexing Method for High-Dimensional Data. In M. A. Nascimento, T. Sellis, R. Cheng, J. Sander, Y. Theng, H.-P. Kriegel, M. Renz, and C. Sengstock, editors, *Advances in Spatial and Temporal Databases: Proceedings of the 13th International Symposium on Spatial and Temporal Databases*, pages 183–200. Springer Berlin Heidelberg, 2013.

[201] R. Weber, H. J. Schek, and S. Blott. A Quantitative Analysis and Performance Study for Similarity-Search Methods in High-Dimensional Spaces. In *Proceedings of the 24th International Conference on Very Large Data Bases*, pages 194–205, 1998.

[202] R. Weeks. Rweeks' R-Tree. URL `https://github.com/rweeks/util`. Last Access: May 4th, 2018.

[203] J. Wegener, A. Baresel, and H. Sthamer. Evolutionary test environment for automatic structural testing. *Information and Software Technology*, 43(14):841–854, Dec. 2001.

[204] J. Wegener, K. Buhr, and H. Pohlheim. Automatic Test Data Generation For Structural Testing Of Embedded Software Systems By Evolutionary Testing. *Proceedings of the 4th Annual Conference on Genetic and Evolutionary Computation*, pages 1233–1240, July 2002.

[205] A. Wert, J. Happe, and L. Happe. Supporting swift reaction: automatically uncovering performance problems by systematic experiments. In *Proceedings of the 2013 International Conference on Software Engineering*, pages 552–561. IEEE Press, May 2013.

[206] N. Williams, B. Marre, P. Mouy, and M. Roger. PathCrawler: Automatic Generation of Path Tests by Combining Static and Dynamic Analysis. In M. D. Cin, M. Kaâniche, and A. Pataricza, editors, *Dependable Computing - EDCC 5: Proceedings of the 5th European Dependable Computing Conference*, pages 281–292, Berlin, Heidelberg, Apr. 2005. Springer Berlin Heidelberg.

[207] M. Woodside, G. Franks, and D. C. Petriu. The Future of Software Performance Engineering. In *Proceedings of the Future of Software Engineering 2007*, pages 171–187. IEEE, 2007.

[208] M. F. Worboys. A Unified Model for Spatial and Temporal Information. *The Computer Journal*, 37(1):26–34, 1994.

[209] P. L. Yu. Cone convexity, cone extreme points, and nondominated solutions in decision problems with multiobjectives. *Journal of Optimization Theory and Applications*, 14(3): 319–377, 1974.

[210] X. Yuan and A. M. Memon. Generating Event Sequence-Based Test Cases Using GUI Runtime State Feedback. *IEEE Transactions on Software Engineering*, 36(1):81–95, 2010.

[211] X. Yuan, M. B. Cohen, and A. M. Memon. GUI interaction testing: Incorporating event context. *IEEE Transactions on Software Engineering*, 97(4):559–574, 2011.

[212] R. Zhang, B. C. Ooi, and K. L. Tan. Making the Pyramid Technique Robust to Query Types and Workloads. In *Proceedings of the 12th International Conference on Data Engineering*, pages 313–324. IEEE Comput. Soc, Mar. 2004.

[213] E. Zitzler, M. Laumanns, and L. Thiele. SPEA2: Improving the strength Pareto evolutionary algorithm. *TIK-Report*, 3242(103):95–100, 2001.

Acknowledgements

First and foremost I thank my supervisors Prof. Dr.-Ing. Elke Pulvermüller and Prof. Dr. rer. nat. Martin Breunig for their guidance and support during the composition of this thesis. I thank Benjamin Graf for the countless helpful discussions about test generation, code coverage and optimization. Next, I thank my master students Niels Meyering, Falk Wilke and Jan-Philipp Schleutker for their work on the predecessors of the code coverage and test generation modules. I thank Dr. Rainer Düsing for the helpful discussions on statistics. For proofreading my drafts I would like to thank Benjamin Graf, Mareike Büscher, Jana Lehnfeld and Judith Katenbrink. I thank Friedhelm Hofmeyer for the continuous, fast and professional support on any technical issue.