# Clustering and Analysis of User Behaviors utilizing a Graph Edit Distance Metric

Bachelor's Thesis

Lars Jürgensen

November 11, 2019

Kiel University
Department of Computer Science
Software Engineering Group

Advised by: Prof. Dr. Wilhelm Hasselbring
Dr.-Ing. Reiner Jung

**Eidesstattliche Erklärung**

Hiermit erkläre ich an Eides statt, dass ich die vorliegende Arbeit selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet habe.

Kiel, 11. November 2019

# List of Acronyms

***GED*** Graph Edit Distance

***PCM*** Palladio Component Model

***OPTICS*** Ordering Points to Identify Clustering Structure

***IBM*** Ideal Behavior Model

***SAM*** Sequence Alignment Method

# Abstract

Understanding the utilization of software systems by users is helpful for utilization forecasts used in performance optimization and improving user experience. Therefore, it is necessary to understand the utilization and be able to identify typical user behaviors. One possibility to find typical behaviors is cluster analysis. Various clustering algorithms, like X-Means, EM-Clustering and BIRCH, were implemented in iObserve. The user actions were transformed into vectors, which were then compared using the Euclidean distance metric. Evaluation showed that these approaches were not able to find all expected behaviors in simple workloads.

We propose an approach, which models the user behaviors as graphs. The Behavior Models are compared using a Graph Edit Distance based distance function. We utilize the M-Tree index structure to make nearest neighbor searches more efficient. The OPTICS algorithm is then utilized to identify the clustering structure of the behaviors.

We implemented our approach in the iObserve project using the pipe and filter architecture. The implementation is evaluated by clustering fixed and randomized workloads of the JPetStore and comparing the clustering results with manually created expected results. The evaluation shows it is possible to detect all expected behaviors in both fixed and randomized workloads.

# Contents

Chapter 1

# Introduction

Understanding the utilization of software systems by users is helpful for utilization forecasts, which can be used for runtime performance optimizations. Knowing the user base can also lead to suitable software evolution, like new features, user interface adaptations and performance improvements. Additionally, it can help creating realistic workload characterizations, which are essential for performance evaluations.

One possibility to find typical behaviors is cluster analysis. Vögele et al. 2018 envisioned the WESSBAS approach to derive workload characterizations. They used the X-Means algorithm to cluster user sessions and find navigational patterns. The user actions were transformed into vectors, which were then compared using the Euclidean distance metric. Later, an approach based on WESSBAS implemented the Expectation–maximization algorithm instead of X-Means. The evaluation showed that both algorithms could not detect all expected behaviors correctly [Jung et al. 2017].

Therefore, further clustering approaches, like BIRCH [Lorenzen 2018] and similarity matching [Jannis 2018], where evaluated. However, these could not deliver better results than X-Means.

The reason for these results might be that the clustering data is transformed into vectors, as this leads to a high amount of dimensions. It is a well known problem, than many clustering algorithms work worse in high-dimensional spaces. This phenomenon is referred to as the curse of dimensionality [Assent 2012].

In order to avoid this problem, we take a different approach. Instead of comparing vectors with a Euclidean distance metric, we transform the user sessions into state transition graphs, and compare them with a Graph Edit Distance (GED) metric. We call these graphs Behavior Models. We then use the density based OPTICS algorithm, to identify clusters in the data set. To avoid a quadratic complexity, we use the M-Tree index structure, which makes more efficient nearest neighbor searches for the OPTICS algorithm possible.

## 1.1 Goals

The central research question of the thesis is to find and evaluate a clustering approach for user behavior models. This goal can be divided into 3 operational goals:

### G1: Find a framework implementing the clustering

The first goal is to identify a suitable implementation or create an usable implementation of the Graph Edit Distance (GED) metric and the M-Tree. The advantage of an existing implementation is that it may contain fewer mistakes, is potentially better maintained and unnecessary extra work is avoided. In case that no suitable public licensed implementation is found, a self-made implementation is used.

### G2: Realize the clustering algorithm in iObserve

In order to integrate this contribution with the existing iObserve software, the corresponding stages will be implemented with the TeeTime pipe and filter framework. Furthermore, using existing stages from iObserve allows to concentrate on the development of the clustering filters and algorithms, as for the behavior model generation existing stages can be reused.

Additionally, we will apply our approach to monitoring data from Jira. Therefore, we will implement a specific Jira event filter. This part is an optional objective, as it highly depends on the quality of the monitoring data and behavior model generation stages.

### G3: Evaluate the implementation

Besides implementing an clustering approach based on graph clustering approaches, we must evaluate its fitness for finding common user behavior patterns. Dornieden [2017] already generated simple monitoring data in the JPetStore. Based on his example workloads, we will (a) use exact user behaviors to test the ability to correctly classify behaviors, (b) create randomized variations of the original behaviors, e.g., buying two instead of one fish, and (c) use real behaviors of students using Jira which where collected during the Software Project course. This last objective is optional and may not yield any substantial results for this thesis.

## 1.2 Document Structure

Chapter 2 will introduce the tools and algorithms we utilize. Our approach and the way it is implemented is described in Chapter 3. In Chapter 4 this approach is evaluated and the results are discussed. Chapter 5 outlines the related work. In Chapter 6 the thesis is finally summarized and future work is suggested.

Chapter 2

# Foundations

In this chapter the foundations of the thesis are introduced. The foundations include the technical and scientific foundation of this thesis. The technical foundations include the frameworks, tools and source code. Whereas the scientific foundation includes algorithms and the research context.

We begin with the TeeTime framework in Section 2.1. It is a pipe and filter framework, which is used in the iObserve project. In Section 2.2, we introduce the Kieker toolset for monitoring. Section 2.3 describes the JPetStore web application, which we use to evaluate our approach. To generate workloads in the JPetStore, we utilize the iObserve Workload Driver. It is portrayed in Section 2.4. We continue with the iObserve project itself in Section 2.5 and the parts of the iObserve behavior analysis, our project is build on, in Section 2.6.

Different behaviors are compared with the Graph Edit Distance (GED) metric. It is introduced in Section 2.7. To make more efficient nearest neighbor queries possible, we store the behaviors in an M-Tree, which is presented in Section 2.8. And finally, in Section 2.9, the clustering algorithm OPTICS is explained, which we use to find clusters.

## 2.1 TeeTime

TeeTime is a generic pipe and filter framework for Java and C++ [Wulf et al. 2017]. The pipe and filter architectural style comprises stages – also called filters – which process data and pipes to connect stages and pass data. The framework enables the concurrent execution of stages and automatically manages the threads for them. In TeeTime multiple stages can be grouped into one composite stage to decrease the complexity of the structure. As iObserve is implemented with TeeTime, the clustering of the user behaviors will be realized using TeeTime stages.

## 2.2 Kieker

Kieker is a framework and toolset to monitor, profile and analyze the run-time behavior of Java applications. This can be visualized in UML sequence diagrams, Markov chains, component dependency graphs, trace timing diagrams, and message and execution trace

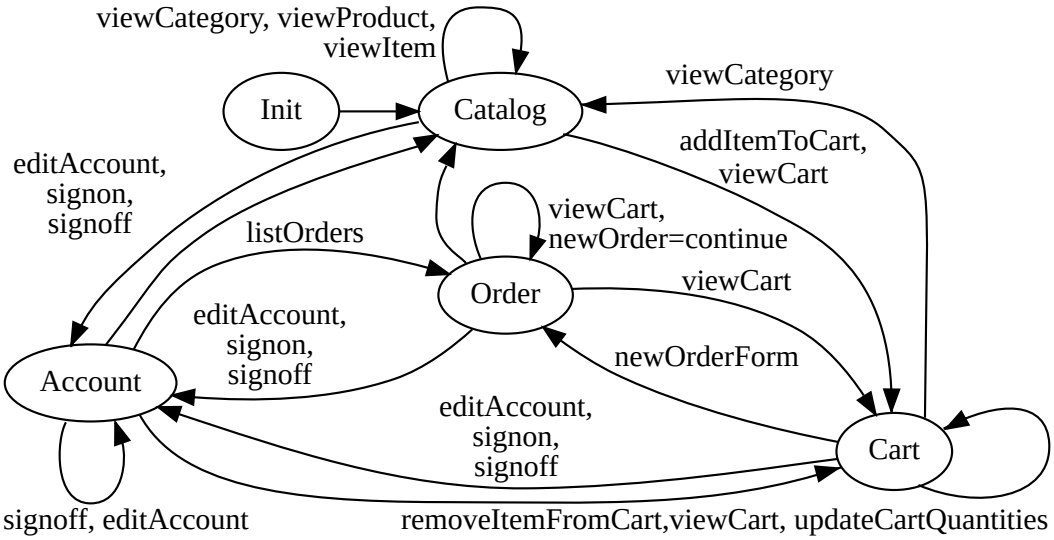


**Figure 2.1.** All servlets and transitions of the JPetStore. Edges with equal source and target servlets are merged into one edge with multiple labels [Jung and Adolf 2018].

models [van Hoorn et al. 2012]. We will use Kieker to monitor the user behaviors to analyze and cluster. The monitoring files can be read by specific stages implemented with TeeTime.

## 2.3 JPetStore

The JPetStore is a simple Java example web application, which is commonly used for research. The application is an online shop for animals. Users can log in, browse products, buy products, change account information etc. Figure 2.1 shows the functionality of the JPetStore in detail. It consist of only 5 different servlets and transitions between those.

The account servlet is used for all account related actions, like creating an account and logging in. The purpose of the catalog servlet is to visit categories, products and items. The cart servlet can be utilized to add and remove items from the cart and to finally buy the items the order servlet is used.

Shimizu et al. 2018 proposed a single service experiment setup of the JPetStore, which we will use to evaluate our approach. The internal structure can be seen in Figure 2.2. Note, that there is a component for every servlet of Figure 2.1.

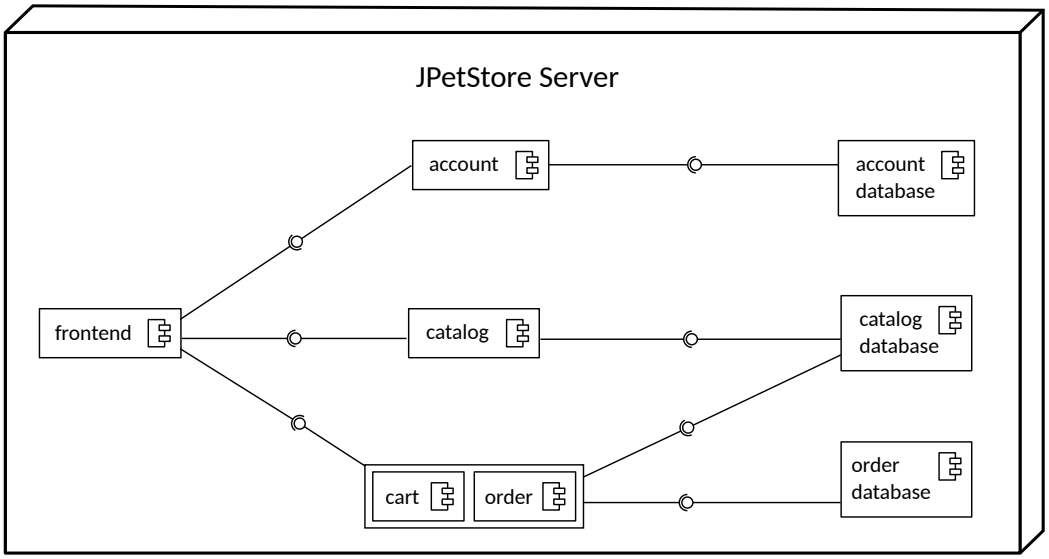


**Figure 2.2.** A component diagram of the JPetStore.

## 2.4 iObserve Workload Driver

Dornieden 2017 created a tool to automatically generate workloads for the JPetStore. It uses the selenium framework, which can automate browser behavior [*Selenium*]. We will use this workload driver to generate user behavior in the JPetStore. The Html Unit Driver is currently used as the webdriver-component, as it is faster than other implementations and no GUI is required for workload generation.

## 2.5 iObserve Project

Software systems are dynamic and change over time. The changes of the software system can be separated into automatic and manual changes. The automatic changes are called adaptations and the manual changes by operators and developers are called evolution. Adaptation and evolution influence each other. This concept is visualized by two interwoven cycles in Figure 2.3.

A key issue of the influence between both cycles is that design time models used to assess software at design time become outdated over time, as runtime changes are not related back to the design. Furthermore, runtime models often lack design information, as they can only refer to implementation artifacts.

iObserve aims to keep the design time models up to date at runtime and enable to use design time model analyses at runtime [Hasselbring et al. 2013]. Furthermore, the use of

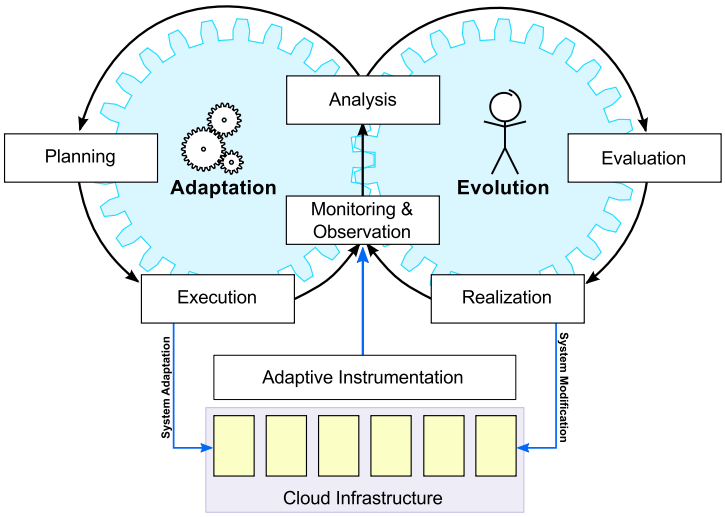


**Figure 2.3.** The iObserve approach [Hasselbring et al. 2013]. The adaptation represents automatic changes from the system itself. The evolution represents changes from the developers and operators. The adaptation and evolution continuously influence each other.

design time models allows operators to understand models better, as they are closer to the conceptual architecture of the software system. Therefore, operators can adjust adaptations and forecasts. For example, operators can inject previously learned behavior models and workload intensities. For this purpose, this thesis concentrates on the generation and analysis of behavior models inside of the iObserve project.

The iObserve project uses the Palladio Component Model (PCM) as an internal representation [Becker et al. 2009], which provides models for the architecture, cost, and user behavior.

The model update and analysis of the iObserve project uses the TeeTime framework to realize its pipe and filter architecture.

## 2.6 iObserve Behavior Analysis

The iObserve Behavior Analysis project contains previous attempts to analyze and cluster user behavior. Part of these approaches was to read Kieker log files, filter for the relevant events and group them by the session. This was implemented using the TeeTime framework. Figure 2.4 shows the TeeTime stages relevant for us and how they are connected. The `Logs Reader Composite Stage` is part of the Kieker framework and can read multiple Kieker logs situated in a common folder.

The `Dynamic Event Dispatcher` selects `Monitoring Records` and forwards them based on their

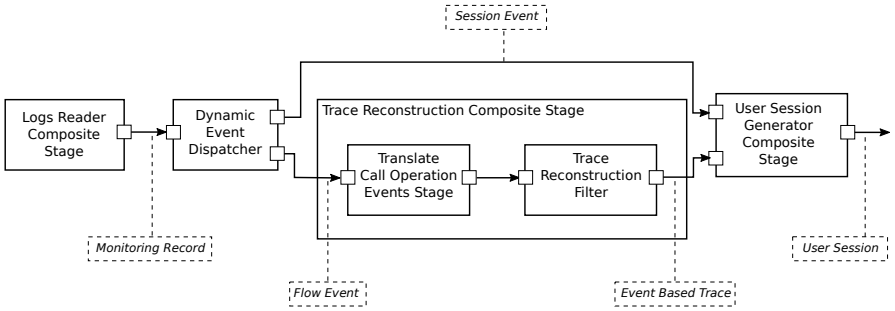


**Figure 2.4.** An overview of the TeeTime stages used to read the Kieker files and transform them into User Sessions.

type.

The Monitoring Records, which represent the start or the end of a session, are forwarded to the User Session Generator Composite Stage. The operations that are caused by users are stored in Flow Events and are processed by the Trace Reconstruction Composite Stage and also forwarded to the User Session Generator Composite Stage.

User Sessions are generated in a five step process shown in Figure 2.5. The entry call stage processes incoming traces to identify entry calls and stores their meta data in Payload Aware Entry Call Event. In this thesis, we will simply refer to them as events. Most of these events are user actions, but some events are caused by other operations like loading a resource. The Entry Call Sequence Filter uses the Session Events, to figure out which events belong to the same session. These events are then stored in a User Session. The goal of the Session Acceptance Filter is to remove unwanted sessions. Currently, all session are accepted, so this stage has no effect. The Trace Operation Cleanup Filter also has no effect, but might later be used, to improve the name of the operations. Finally, the User Session Operation Cleanup Stage removes unwanted events from the user sessions. In the JPetStore these unwanted events are those, which are caused by loading images and stylesheets.

## 2.7 Graph Edit Distance

The Graph Edit Distance (GED) is a distance metric for graphs [Sanfeliu and Fu 1983]. The idea is, to compare two graphs, by transforming one graph into the other. The distance between the graphs then correlates to the complexity of the transformation, as similar graphs as easier to transform into another. The allowed set of operations for the transformations can be determined based on the context. Typical operations are:

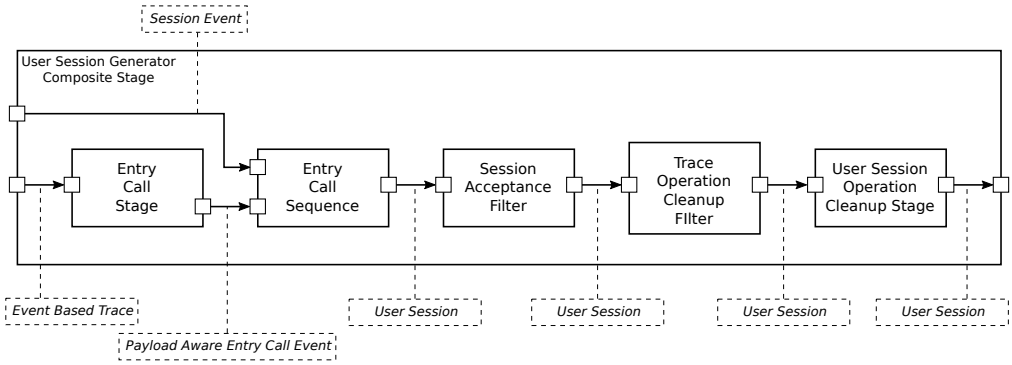


**Figure 2.5.** The detailed structure of the User Session Generator Composite Stage.

▻ edge/node insertion

▻ edge/node deletion

▻ edge/node label substitution

The cost function $c$ gives each transformation $e$ the cost $c(e)$. For two graphs $g_1$ and $g_2$ is an edit path a set of transformations that transforms $g_1$ in $g_2$. The set of all edit paths is denoted as $\gamma(g_1, g_2)$. The GED is defined as follows:

$$GED(g_1, g_2) = \min_{e_1,\ldots,e_k \in \gamma(g_1,g_2)} \sum_{i=1}^{k} c(e_i)$$

In words, the GED is the minimal total cost of all edit paths, between two graphs. We will use a GED metric, to compare different behaviors. An example is shown in Figure 2.6.

## 2.8 M-Tree

The M-Tree is a data structure, which stores objects based on a distance function $d$ [Ciaccia et al. 1997]. In comparison to multi-dimensional access methods, the M-Tree does not need an absolute position in metric space, only a relative distance between objects. This is an advantage, because many metrics – like the GED – do not work with absolute positions.

The M-Tree is, as the name indicates, a tree. There is a minimum and a maximum amount of entries a node of the M-Tree can contain. A leaf node only contains entries of the following structure:

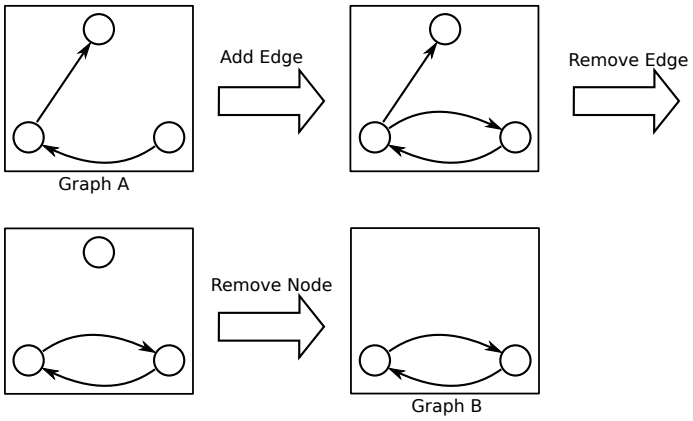


**Figure 2.6.** A GED example. The distance between graph A and B is 3, when all operations have a cost of 1.

| | |
|---|---|
| $O_j$ | the object, this entry represents |
| $d(O_j, P(O_j))$ | distance to parent of $O_j$ |

There is an entry like this for every object $O_j$ that should be stored in the M-Tree. The internal non leaf nodes have the function to organizes the nodes in a way that close objects are stored together. For this purpose, the internal nodes have entries with routing objects:

| | |
|---|---|
| $O_r$ | the routing object |
| $ptr(T(O_r))$ | pointer to sub-tree $T(O_r)$ |
| $r(O_r)$ | covering radius of $O_r$ |
| $d(O_r, P(O_r))$ | distance to parent of $O_r$ |

These entries contain an object and the distance to its parent, just as the entries of leaf nodes. But additionally they have a pointer to a sub-tree $T(O_r)$ and a covering radius $O_r$. The covering radius is the maximal distance to $O_r$ any object in the sub-tree can have, so that only close objects are contained in the sub-tree.

The distance function $d$ has to satisfy three criteria:

$$d(O_x, O_y) = d(O_y, O_x) \text{ (symmetry)} \tag{2.1}$$

$$d(O_x, O_y) > 0, if(O_x \neq O_y), \text{ and } d(O_x, O_x) = 0 \text{ (non negativity)} \tag{2.2}$$

$$d(O_x, O_y) \leqslant d(O_x, O_z) + d(O_z, O_y) \text{ (triangle inequality)} \tag{2.3}$$

Due to these constraints for the function $d$, the nearest neighbor queries are more efficient. Nearest neighbor queries on an object either search all objects within a specified range or the k nearest neighbors according to the distance function $d$. On unsorted objects

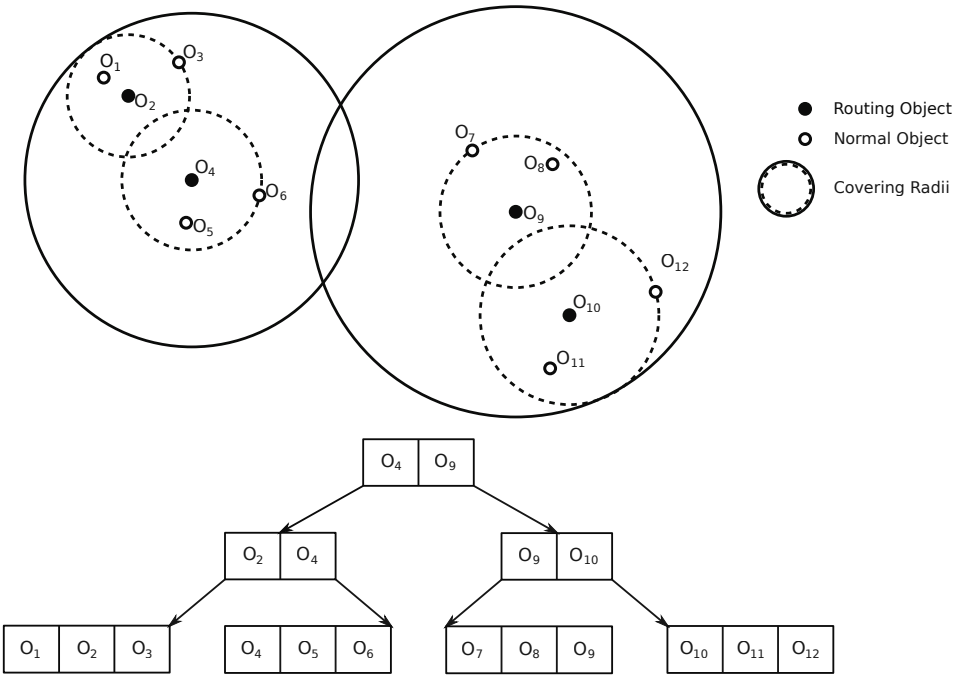


**Figure 2.7.** An M-Tree example with objects in 2D space.

this takes $\mathcal{O}(n)$ distance calculations, as all objects have to be checked. When the objects are stored in an M-Tree, only $\mathcal{O}(log(n))$ distance calculations are needed, when the k or the search range is small [Mahapatra and Chakraborty 2015]. The complexity decreases, because the routing objects helps finding the closest objects.

The M-Tree has an insert method to add new objects and a split method in case nodes contain too many entries. This makes it a balanced tree. Figure 2.7 shows an example of an M-Tree in 2D space.

## 2.9 OPTICS

The Ordering Points to Identify Clustering Structure (OPTICS) algorithm is a density based algorithm to analyze the cluster structure of a data set [Ankerst et al. 1999]. The OPTICS algorithm does not create clusters, but creates a *reachability-plot*. Two methods to extract clusters from the *reachability-plot*are later introduced.

The OPTICS algorithm is based on the DBScan algorithm [Ester et al. 1996] and inherits the two parameters $\epsilon$ and *MinPts*. The *MinPts* parameter determines how many neighbors an object requires, to be in a dense region. The $\epsilon$ parameter is the maximal distance two

neighbors can have. We will first introduce the DBScan algorithm, as it makes the OPTICS algorithm more comprehensible.

The objects, which have at least *MinPts* neighbors in a radius $\epsilon$, are called *core-objects*. All objects, which have a *core-object* as a neighbor, are called *directly density-reachable*. The *core-object* is only a neighbor, if the distance is smaller than $\epsilon$. Intuitively, this means that *core-objects* are within a dense region and *directly density-reachable* objects close to a dense region.

An object $p$ is *density-reachable* from another object $o$ , when there exists a path of objects from $p$ to $o$, so that every object is *directly density-reachable* from the previous object. This means that *density-reachable* is the transitive hull of *directly density-reachable*.

Two objects are *density-connected*, when both objects are *density-reachable* from the same object. Intuitively, all objects inside or close to the same dense region are *density-connected* with each other. A cluster is defined as a maximal set of *density-connected* objects. The objects, which do not belong to any cluster are treated as noise.

One of the main weaknesses of the DBScan algorithm is detecting clusters with different densities. This is because the parameters define the required density for a cluster. Thus, clusters with varying densities hinder the detection.

The OPTICS algorithm tries to solve this problem, by returning a *reachability-plot* instead of clusters. In the DBScan algorithm the higher the $\epsilon$ values is, the more objects are assigned to clusters. The OPTICS algorithm instead determines for each object the minimal $\epsilon$ value, which is necessary for an object to be in a cluster. This minimal $\epsilon$ value is referred to as $\epsilon'$. The $\epsilon$ parameter of the OPTICS algorithm is a limit for the $\epsilon'$values. Therefore $\epsilon' \leqslant \epsilon$ is always satisfied.

OPTICS assigns the objects a *core-distance* and a *reachability-distance*. The *core-distance* of an object is the minimal radius $\epsilon'$around it, in which there are at least *MinPts* objects. In other words, it is the $\epsilon'$ value, from which on the object would be a *core-object*. The *reachability-distance* of an object $p$ with respect to another object $o$ is the smallest $\epsilon'$ value, such that $p$ is *directly density-reachable* from $o$ if $o$ is a core object. Both, the *core-distance* and the *reachability-distance* are *UNDEFINED*, if the results exceed $\epsilon$. Thus, $\epsilon$ can be seen as a limit for the neighbor search distance.

The algorithm visits the objects successively. In the process, nearby objects are assigned the minimal *reachability-distance* with respect to all previously visited objects. Objects with a low *reachability-distance* are the prioritized next objects to visit, so close objects are visited together. When an object is visited, it is added to the plot in combination with the minimal *reachability-distance* to the previous objects. Intuitively, each object is assigned the $\epsilon'$ value, from which on the object would be part of a cluster. This means an object with a very small *reachability-distance* is part of a very dense region and an object with a higher *reachability-distance* is part of a less dense region. Therefore, in dense regions, the *reachability-distance* values of the *reachability-plot* are smaller. This leads to valleys, when the *reachability-plot* is graphically depicted. Thus, this depiction can be manually interpreted, as these valleys correspond to clusters. An example of the *reachability-plot* can be seen in Figure 2.8.

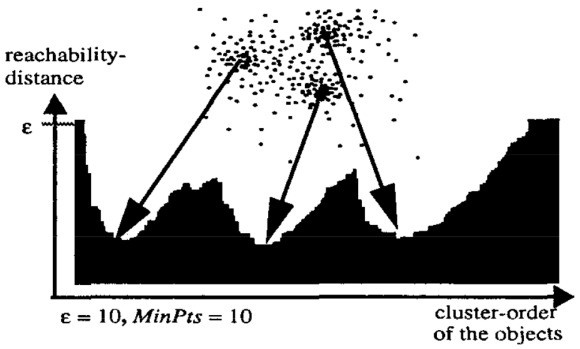


**Figure 2.8.** A *reachability-plot* example from the OPTICS algorithm. The dots represent the objects to be clustered. The valleys of the *reachability-plot* correspond to these clusters [Ankerst et al. 1999].

There are also several automatic ways to extract the clusters from the *reachability-plot*. An simple method is to choose an $\epsilon'$ value and let all valleys below this value be a cluster. These clusters are – except for some corner cases – equivalent to the result of the DBScan algorithm [Ankerst et al. 1999].

Alternatively, we could extract hierarchical clusters, called $\xi$-clusters [Ankerst et al. 1999]. The concept behind $\xi$-clusters is to first detect potential start and end points of clusters and then combine these to actual hierarchical clusters. The start and end points are regions, where the *reachability-plot* rapidly increases or decreases. This method is able to overcome the problem with varying densities, as clusters can contain further clusters with higher densities.

The complexity of the OPTICS algorithm depends on the complexity of the nearest neighbor queries. Nearest neighbor queries either return the k closest neighbors or all neighbors within a range of $\epsilon$. The amount of required queries of the OPTICS algorithm scales linear with the amount of objects. Without an index structure, a nearest neighbor query has a complexity of $\mathcal{O}(n)$, as all objects are potential neighbors. This leads to an overall complexity of $\mathcal{O}(n^2)$. Tree based index structures – like the M-Tree – can do nearest neighbor queries in $\mathcal{O}(log(n))$, so the complexity of the OPTICS algorithm is reduced to $\mathcal{O}(n * log(n))$.

The authors observed that the shape of the *reachability-plot* is mostly independent from the parameters. The lower the $\epsilon$ parameter is, the more objects have a *reachability-distance* of *UNDEFINED*, so clusters with lower densities can not be seen anymore. The only disadvantage of a high $\epsilon$ parameter is that the algorithm slows down. For example with an $\epsilon$ value of infinity, the nearest neighbor queries always return the entire database, as all objects are within that distance [Ankerst et al. 1999]. However, there is a large range of $\epsilon$ values, where the shape of the *reachability-plot* does not significantly change. The *MinPts* parameter also has a limited effect on the shape of the *reachability-plot*Ȯnly *MinPts* values

below 10 can cause a jagged plot, which leads to many small clusters [Ankerst et al. 1999].

Chapter 3

# Clustering Approach

This chapter will explain the approach to analyze and cluster the behaviors and describe how it was implemented. The basic idea is outlined in Section 3.1, the structure of the project and the TeeTime stages is described in Section 3.2. Our approach to model the user behaviors is presented in Section 3.3. In Section 3.4 we explain the way we compare the Behavior Models with the GED. We explain the implementation of the M-Tree in Section 3.5 and the implementation of the OPTICS algorithm in Section 3.6. Section 3.7 finally describes, how we find a representative objects for each cluster.

## 3.1 Concept

The conceptual architecture describes the basic building blocks of our clustering approach. Figure 3.1 shows the conceptual architecture of the approach. Instead of transforming the behaviors into vectors, as previous approaches did [Dornieden 2017; Jannis 2018; Lorenzen 2018], we generate a graph with a hierarchical structure for each behavior. We compare these graphs with a Graph Edit Distance (GED) (cf. Section 2.7). This metric is used to index the behaviors in an M-Tree, where similar behaviors are grouped together. The next step is to apply the OPTICS algorithm on the behaviors. It uses the M-Tree to find nearest neighbors. This reduces the complexity of the OPTICS algorithm from $\mathcal{O}(n^2)$ to $\mathcal{O}(n * log(n))$, as described in Section 2.9. It also makes use of the GED metric. The result is the reachability plot (see Section 2.9). This plot already makes the clustering structure visible. However, we use a simple algorithm to extract the clusters from the OPTICS result, as the main goal is to generate actual clusters.

## 3.2 Analysis Structure

The entire analysis is realized using the TeeTime framework. An overview of the used stages can be seen in Figure 3.2. The Kieker monitoring data is read and transformed into User Sessions using the analysis stages of the previous behavior analysis projects described in Section 2.6. The stages are all part of the Model Generation Composite Stage, but are represented by the User Session Generation stage, to keep the diagram clear. The User Sessions are then transformed into Behavior Models, which is described in Section 3.3. The Behavior Models are then send to the Clustering Composite Stage and the generated clusters are

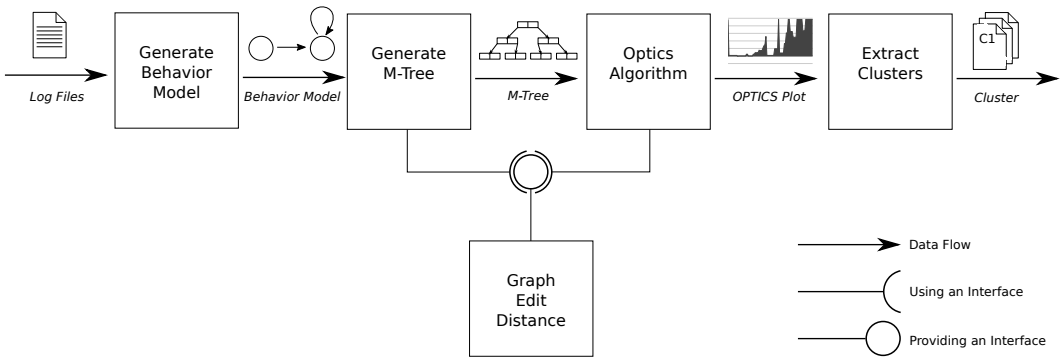


**Figure 3.1.** The clustering concept. The Log files are in multiple steps transformed into behavior clusters. The acsged is used by the M-Tree generation and the OPTICS algorithm.

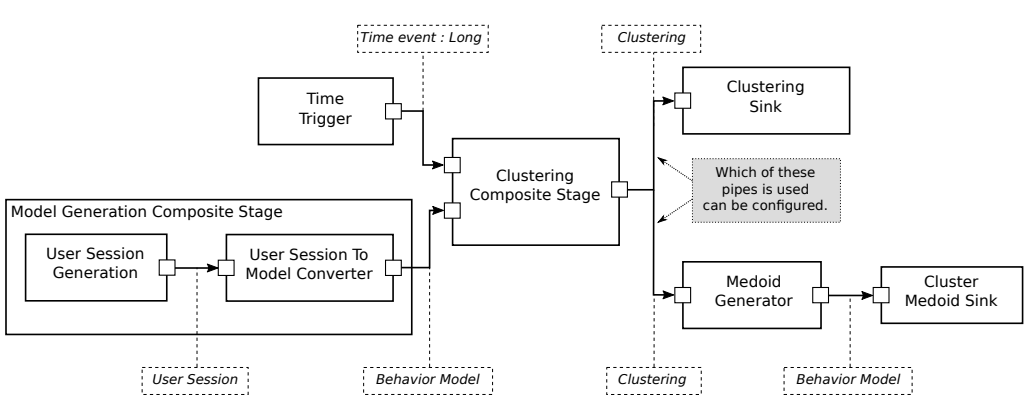


**Figure 3.2.** Overview of the TeeTime configuration of the clustering approach.

stored in a data type, called Clustering. The Time Trigger Stage is used to initialize the cluster generation. The Clustering can either be send to the Clustering Sink or to the Medoid Generator. The Clustering Sink writes all clusters including the behaviors, which were classified as noise, into a file. The Medoid Generator calculates a representative behavior model for each cluster and sends these models to the Cluster Medoid Sink. These representative models are subsequently written into different files labeled with the cluster numbers.

The composition of the Clustering Composite Stage is depicted in Figure 3.3. The received behavior models from the User Session To Model Converter are send to the Behavior Model To Optics Data Converter, which stores the Behavior Models in the Optics Data objects. Optics Data

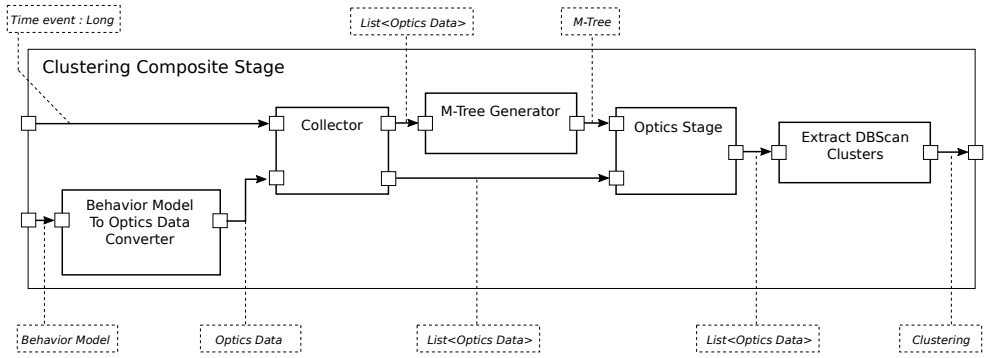


**Figure 3.3.** The Clustering Composite Stage. The above input port is a Time Trigger Stage.

is just a wrapper class for Behavior Models, which assigns each model 3 additional attributes. The attributes are the *core-distance*, the *reachability-distance* and a boolean, which states, whether or not the object has been visited before. They are further described in Section 2.9.

The Optics Data objects are send to the collector. The collector stores all received models, until either it is triggered by the timer or a maximum value of received models is reached. Then all new models are send in a list to the M-Tree Generator and the Optics Stage. The M-Tree Generator creates an M-Tree, which contains all models from the list. The Optics Stage now receives both, the list of models and the M-Tree. The result of the algorithm is an ordered list of Optics Data where each has an *reachability-distance* value. The final stage extracts the clusters from the list and stores them in a Clustering object.

## 3.3 Behavior Model

Comparing two behaviors by directly comparing the list of events is very complex. Therefore, we want to structure the behaviors in a way that simplifies finding similar or equal parts. The relevant attributes of an event are the operations signature, the parameter names and the parameter values. An example User Session with 7 events can be seen in Table 3.1. We will use this example to explain the structure of the Behavior Models.

The basic structure of the Behavior Model is similar to that of a state transition graph. As the operations signature correlates to the servlet of the application that is used, it can be seen as a state. Thus, we use the operation signatures as the nodes of the graphs. The transitions between different states are caused by events. So we allocate the events in directed edges, which connect the nodes. We artificially add one *Init* node to the graph. We define this node as the source of the edge with the first event, as no previous operation signatures exists. It represents the start of the session.

**Table 3.1.** A simplified example User Session with 7 events. Note that the order of the events is important. This is an JPetStore example, where the user selects a cat, adds it to the cart and logs in afterwards.

| Number | operations signature | parameter names | parameter values |
|---|---|---|---|
| 0 | Catalog | [] | [] |
| 1 | Catalog | ["viewCategory"] | ["CATS"] |
| 2 | Catalog | ["viewProduct"] | ["FL-DLH-01"] |
| 3 | Catalog | ["viewProduct"] | ["FL-DLH-02"] |
| 4 | Cart | ["addItemToCart","workingItemId"] | ["","EST-16"] |
| 5 | Account | ["signonForm"] | [""] |
| 6 | Account | ["username","password"] | ["max","123456"] |

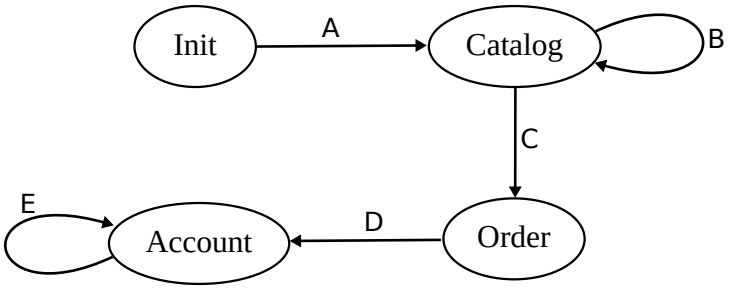


**Figure 3.4.** An example of a Behavior Model. The inner structure of an edge is not depicted. The nodes can be seen as the states of the user session and the edges as the transitions.

Figure 3.4 shows an example Behavior Models based on the events from Table 3.1. The labels from A to E are only used to simplify the explanation and are not part of the actual Behavior Models. The events 1, 2 and 3 are all allocated to the edge B, as they all belong to the transition from Catalog to itself. Event 2 and 3 are very similar, as both events represent visiting a product. However, event 1 is less similar to them, as it represents visiting a whole category. This can be seen in the parameter names. It was observed that different user actions can be distinguished by the parameter names in general. Thus, we group all events of an edge with equal parameter names together, to highlight the types of user actions. We call these groups event groups.

The class diagram of the Behavior Model implementation can be seen in Figure 3.5. The models stores the nodes in hashmaps with their operation signatures as the key. Therefore, it can easily be determined, whether different models have equal nodes. For the same reason, nodes store the ingoing edges with their source node as a the key.

The sink stages of the TeeTime configuration write the Behavior Models as JSON objects. During the serialization, some attributes like the entry time are omitted for the sake of clarity. Currently, all of the omitted fields have no impact on the cluster analysis. In case

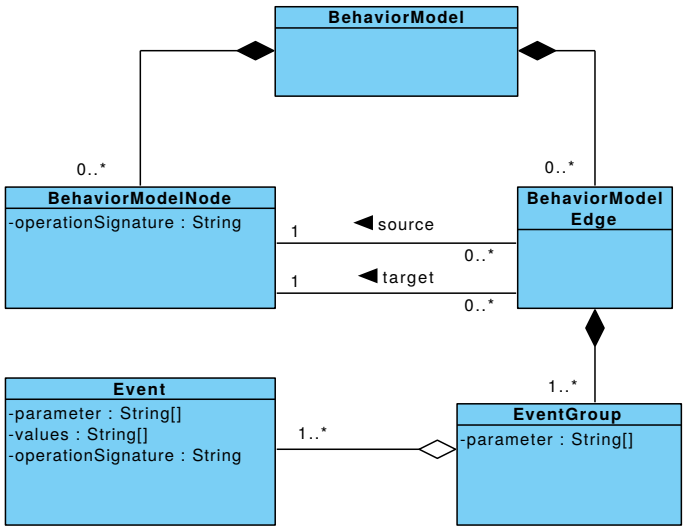


**Figure 3.5.** Class diagramm for the Behavior Model graph

these fields are relevant in future scenarios, they can be added to the graph by adjusting the event serializer.

Some references, which lead to circular references, are also omitted during the serialization. Otherwise the serialization does not terminate, as the JSON objects become infinity large. The deserialization is implemented manually, in order to retrieve these references. Due to the hashmaps, the referenced objects can be found in an efficient way.

## 3.4 Graph Edit Distance

As we want to use the M-Tree to index the models, the distance metric has to satisfy the symmetry (cf. Equation 2.1 ), non-negativity (cf. Equation 2.2 ) and triangle-inequality (cf. Equation 2.3 ) properties. In order to satisfy the non-negativity property, we define all operation costs positive. Thus, no negative distances are possible. We define the costs of insert and delete operations of an object always as equal, so the symmetry property is satisfied. And the triangle inequality is satisfied, because, if there is are paths from $O_x$ to $O_z$ and from $O_z$ to $O_y$, the GED from $O_x$ to $O_y$ can never be greater than the cost of those paths combined, as the combined path would have a lower cost [Sanfeliu and Fu 1983]. This makes it a qualified distance function.

We defined the GED metric without node label and edge label substitutions. Instead we added operations, to add and delete event groups and events. Also, we defined two

**Table 3.2.** The GED operations. The weighted costs differs, depending on the content of the event.

| Object | Operation | Cost |
|---|---|---|
| Node | Insert/Delete | 80 |
| Edge | Insert/Delete | 30 |
| Event Group | Insert/Delete | 10 |
| Event | Insert/Delete | weighted |
| Duplicate Event | Insert/Delete | weighted |

operations to add and delete duplicate events. These are events, where both the parameter names and the parameter values are identical. We separated them from the normal event operations, because the repetition of identical events conveys less information about the user behavior.

The insertion cost should always be greater or equal than the duplication cost. If the duplication cost were greater, the operation would never be used, as the minimal operation path is determined.

The list of operations and their costs can be seen in Table 3.2. We gave operations of significant objects like nodes higher costs than operations of objects like event groups. Behavior Models typically have way more event groups than Behavior Models. If the operation costs for event groups were as high as the operation costs for nodes, the impact of the nodes to the distance would be minor. Other than that, the costs are arbitrarily chosen. These costs can be configured using a configuration file. All configuration are shown in Appendix A.

The last challenge is the comparison of the parameter values of the events. All events in one event group have the same source and target and the same parameter names. But sometimes the parameter values add a lot of information to the distance and sometimes none. In previous approaches the values were only compared completely manually [Jannis 2018]. Equal items have a distance of 0, different items of the same product a distance of 1, different products of the same category a distance of 2 and so on.

The difficulty is that importance of the parameter values highly depends on the context. We show this with two examples:

```
Event A:
parameterNames: [ "username", "password", "signon"],
parameterValues: [ "max", "1234", "Login"],

Event B:
parameterNames: [ "username", "password", "signon"],
parameterValues: [ "erika", "asdf", "Login"],
```

Although the parameter values of event A and event B are different, the distance should be 0, as the performed action is the same and the username and password are irrelevant for the analysis of user behavior.

```
Event C:
parameterNames: [ "viewCategory", "categoryId" ],
parameterValues: [ "", "CATS" ]

Event D:
parameterNames: [ "viewCategory", "categoryId" ],
parameterValues: [ "", "FISH" ]
```

The events C and D should, in contrast, have a relatively high distance, as it might be interesting, which of the categories fish and cats is used more or in what way.

Our solution to this problem is have a weight function $weigth : Events \rightarrow \mathbb{R}$, which defines the importance of the attribute values. The insert/deletion cost of an event is defined by this function. This makes it is possible to define the weight function in a way that the weight is low, if the event contains the parameter "username", and high, if the event contains the parameter "categoryId". This weight function has to be defined individually for each application. In the JPetStore, there are more examples like an address formula, which should have a low weight and a view product event, which should have a weight a little lower than the category example. In general it makes sense, to give all events, which contain parameter values like personal information or cookies a weight of zero.

It is important to note that the weight of zero does not mean that the event is completely ignored. The event can cause an event group to exist, which might increase the distance between two sessions. This effect is intended, because it is relevant, whether a user is logged in, but not which password the user has. If one event should be completely ignored, the User Session Operation Cleanup Stage can be used (cf. Section 2.6).

A similar decision is necessary when comparing duplicate events. Sometimes the same event occurs multiple times and it depends on the context, whether the amount is relevant. It is for example a bigger difference, how often someone buys a product that how often someone views a product. This also depends on the application and the goal of the analysis, so there is another weight function, which defines the duplication costs of specific events.

The GED between two behavior models is realized in a class which implements the DistanceFunction from the M-Tree project (see Section 3.5). In general, the GED is a Turing-Complete problem, as a lot of operation combinations must be tried out. There are more efficient approximation solutions, but with our specific definition, we can get an efficient and exact solution.

As we do not allow node and edge substitution operations, we know which operations are necessary. This is because all objects that appear in one, but not both of the models have to be inserted or deleted. In this context two nodes are equal, if and only if (iff) their operations signature is equal. Two edges are equal, iff the source and target nodes are equal and two event groups are equal, iff they belong to equal edges and the parameter names are equal.

A naive solution to check, if an equal object exists in the other model is to compare

all pairs. However, this leads to $\mathcal{O}(n^2)$ comparisons in total. A more efficient solution is possible by using the hashmaps: As the nodes are stored in hashmaps, an equal node can be found in $\mathcal{O}(1)$, which lead to $\mathcal{O}(n)$ overall comparisons. Equal edges can be found in a similar way, as they are also stored in hashmaps. Although event groups can be implemented in a similar way, we simply compare all pairs, as an edge most times only contains 1-2 event groups in our experience.

The *weight* function is solved with the interface Parameter Weighting. This interface has two methods: one for the insertion and one for the deletion cost of an event.

We created the Parameter Weighting implementation JPetStore Parameter Weighting, where the insertion cost of login and address formula events are zero, as the differences of the values are irrelevant for the analysis. The category event insertion cost is increased, as it conveys more information about the behavior.

## 3.5 M-Tree

Our M-Tree implementation uses a Java implementation by D'Avila 2013. We integrated his code into the iObserve project using the Gradle build system. The implementation is generic and offers an interface DistanceFunction with a calculate method, to calculate the distance between two generic objects. The GED is realized in a class, which implements this interface. Thus, the M-Tree can use the GED as its distance function.

The only public methods of the M-Tree are add (to add an object to the tree), remove (to remove an object) and getNearest (to find the nearest neighbors of an object). The internal structure of the M-Tree can not be accessed, so this implementation can only be used to improve the efficiency of neighborhood searches.

The non-negativity property of the distance function states that two objects should always have a distance greater than 0. The documentation of the M-Tree implementation also warns that the state of the M-Tree is undefined, when two equal objects are added. There are often behavior models in the JPetStore data set, which have a distance of 0 with our GED definition.

One way to solve this would be, to increase all GED distances by a fixed value, so a distance of 0 is not possible. This is counter intuitive, as objects would have a positive distance to themselves. Also, the non negativity property in Equation 2.2 would be dissatisfied. Thus, the GED would no longer be a metric and still not satisfy all criteria for the distance function of an M-Tree .

Another solution is to preprocess the data, by searching equal Behavior Models and only use one proxy Behavior Model for the clustering. This would even increase the efficiency of further algorithms, as less Behavior Models have to be clustered. However, this creates another problem. The OPTICS algorithm largely depends on the amount of neighbors an object has. So, when only the proxy Behavior Models are considered, the results are falsified, as the Behavior Models have less neighbors. To solve this, the nearest neighbor queries have to consider the amount of Behavior Models one proxy represents. This could be done by

putting the M-Tree in a wrapper class. The M-Tree itself then only stores the proxy Behavior Models, but the wrapper class corrects the results by adding the Behavior Models a proxy represents back in.

None of these solutions are currently implemented, as the M-Tree still delivered reasonable results and no disadvantages were noticed. However, the implementation of a solution might increase the quality and safety of the code.

## 3.6 OPTICS

The existing OPTICS implementations we found did not match our requirements. In the JSAT implementation the distance metric has to be vector based. As our approach is to represent the sessions in Behavior Models, this is not suitable. The WEKA implementation clusters objects of the type Instance. These Instances can have a set of attributes, but no hierarchical or graph based structure. They avoided an object oriented approach to increase the performance. In the ELKI implementation, any type can be used. However, in ELKI, you have no control over what index structure to use. As we decided to use an M-Tree, a self-made implementation used.

The implementation of the OPTICS algorithm followed the paper of Ankerst et al. 1999. The algorithm assigns the Behavior Models the additional attributes *core-distance* and *reachability-distance* (see Section 2.9). This was solved by creating the wrapper class Optics Data for Behavior Models. The Optics Data objects also remembers, whether or not they were visited before. The distance function for Optics Data is defined by the distance between the contained Behavior Models. Visiting the objects with the lowest *reachability-distance* distance first, is realized utilizing a priority queue. This priority queue uses a custom comparator, which compares two Optics Data objects by comparing their *reachability-distance*. The result of the OPTICS algorithm is a sorted list of Optics Data objects. This is enough, as the Optics Data objects each contain a Behavior Model and the associated *reachability-distance*.

As mentioned in Section 2.9, there are multiple ways to automate the extraction of the clusters. We implemented the extraction of DBScan clusters, as it is easier to implement. This is realized in the Extract DBScan Clusters Stage. The clusters are then stored in a Clustering object. The Clustering objects contain a set of clusters and a set of noise objects. These are the objects that could not be classified into a cluster.

## 3.7 Medoid

The clustering results of large data sets often are too incomprehensible for manual analysis. One way to simplify the results is to generate representative objects for each cluster. Jannis 2018 implemented two approaches: In the first approach the representative object has all nodes and all edges any element of the cluster has. In the second approach, only the nodes and edges are picked, which all of the cluster elements share. The problem with both of

these approaches is that they are highly sensible for outliers. A solution to these challenges is choosing the medoid of a cluster. The medoid of a set is the element, with the least average distance to all other elements [Struyf et al. 1997].

The naive approach to find the medoid is to compare all pairs of models, which has a complexity of $\mathcal{O}(n^2)$. We implemented the trimed algorithm instead, which is a slight run-time improvement. The algorithm uses the fact that some elements can be ruled out, due to the triangle-inequality. Since the GED metric satisfies this, the exact medoid is still found [Newling and Fleuret 2016].

Chapter 4

# Evaluation

We evaluate the fitness of our user behavior clustering approach utilizing the JPetStore application with proper workloads. The JPetStore application is a concise yet complete shop system representing one widely used type of web application. The workloads where designed to cover all areas of the application. The experiment setup is explained in Section 4.1 and the results are described and discussed in Section 4.2. Our efforts towards detecting clusters within the Jira logs is documented in Section 4.3.

## 4.1 Experiment Setup

This section explains the experiment setup. The concept behind the evaluation is depicted in Figure 4.1. We generate fixed and randomized workloads in the JPetStore. They are clustered with our approach. The results are than compared to manually generated ideal results.

### Workload Generation

Dornieden 2017 defined 7 user types with predefined tasks. These user types cover all aspects of the application. The user types are all unique. However, they often share sub graphs or parameters with other user types or the entire structure is identical. Therefore, it is not sufficient to consider the parameters or the structure alone. These user types are:

*Account Manager* The Account Manager logs in to the JPetStore and changes his account information. He then views his order.

*Browsing User* The Browsing User views one product in the CATS category and one product in the FISH category.

*Cat Lover* The user Cat Lover enters the store, visits the category CATS and adds 9 cats to the cart. He then signs into the shop, proceeds to the checkout and creates a new order.

*Fish Lover* The user Fish Lover behaves as the Cat Lover, but buys fish instead of cats.

*New Customer* The user New Customer creates a new account. He then copies the behavior of the Single Reptile Buyer, so he visits the REPTILES category and buys one product.

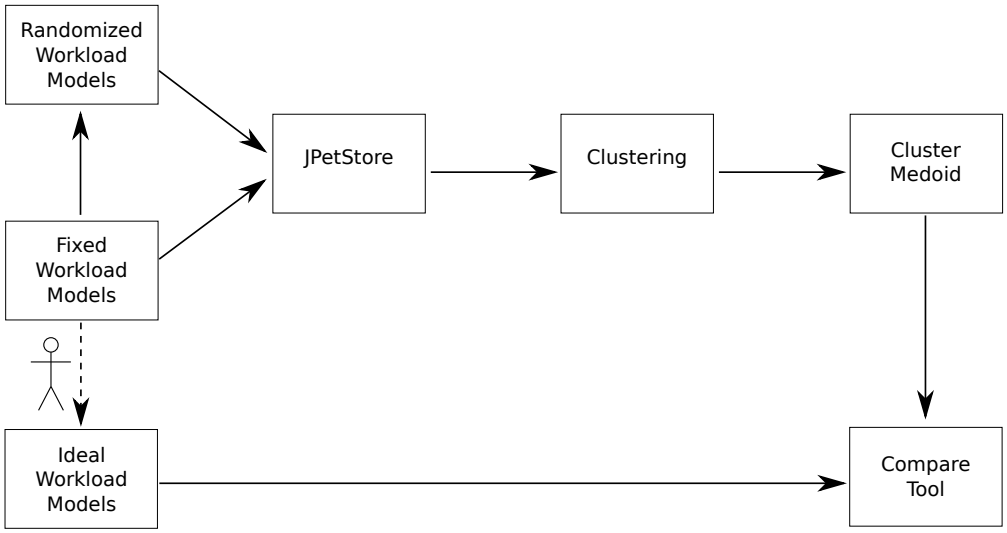


**Figure 4.1.** The concept of the evaluation. We create fixed and randomized workload for the JPetStore. The clustering results are compared to manually generated Ideal Behavior Models

*Single Cat Buyer* The Single Cat Buyer visits the CATS category, adds one product to cart and proceeds to checkout.

*Single Reptile Buyer* The Single Reptile Buyer visits the REPTILES category, adds one product to cart and proceeds to checkout.

Based on the user types above we created different workload profiles consisting of behaviors and intensities. The intensities reflect how often a behavior is triggered in a certain time. As all behaviors of a workload profile are identical, we refer to them as fixed behaviors.

In detail, we created one workload profile for each user type and one combined profile with all behaviors. As intensity, we used a constant model which produced one new user behavior every 10 seconds, resulting in 11 complete user behaviors for each behavior model. Thus, 7 clusters with each 11 behaviors are expected.

As real workloads normally do not contain a lot of identical workloads, the clustering algorithm should also be able to group similar behaviors, which are not necessarily identical.

To be able to evaluate our clustering approach, we created randomized behaviors. This is realized by randomly varying the number of tasks performed by each behavior. In addition, we could have varied some of the parameters, like buying different types of cats or buying different numbers of fish. This was not realized, as we had issues with the workload generation.

The issues with the iObserve workload driver resulted in incomplete and broken workload profiles. We were able to fix most of them (cf. Appendix B). Unfortunately, more problems occurred while generating randomized behaviors. Therefore, we generated the behaviors for each workload profile separately. These 7 workloads were then analyzed together. We created 41 behaviors of each user type. The increased amount of behaviors lead to a higher density, which makes clustering easier. Due to another workload generation issue, the New Customer behavior was not randomized at all. Instead, only 11 identical behaviors were generated.

## Ideal Behavior Models

In order to check, if the clustering results are as expected, we needed to define Ideal Behavior Models (IBMs) (cf. Section 3.6), for each user type. This medoid should be represented with the new Behavior Model data type. For the fixed user behaviors, the cluster medoid could be any of the behaviors, as all should be identical. The first attempt was to use the IBMs from Dornieden 2017 and convert them into Behavior Models. He stored parameter data with the nodes of the behavior graph. This made it unclear, which edge the parameter data would belong to in our Behavior Models. Additionally, it was observed that some nodes had more outgoing than ingoing edges. This is only possible in the initial node, but occurred also for other nodes, which indicates errors in the previous IBMs. Therefore, we decided to design the IBMs from scratch.

We generated Kieker monitoring data of one user type and filtered out the Entry Level Before Operation Events, which represent a user action. The Entry Level Before Operation Events of one session was compared to the workload profile definition and the iObserve Workload Driver project. An overview of the events that correspond to each task can be seen in Appendix C. These were than used to manually create the IBMs.

## Analysis of Result Qualities

After the clustering, we compared the clustering results with the actual IBMs. If each clustering medoid matches with one of the IBMs, we consider the clustering results to be a fitting.

This tool was realized in an iObserve project using TeeTime. An overview of the TeeTime configuration can be seen in Figure 4.2. The Model Reader Stages read JSON Behavior Models. One of the Model Reader Stages reads an IBM and the other one reads a Behavior Model from the clustering results. The Model Comparison Stage takes these Behavior Models and compares them. The comparison results are stored in the Behavior Model Difference class. This class stores all missing and additional nodes, edges, event groups and events and the GED. There exist two sink stages for this configuration: The Comparison Output Stage writes the entire Behavior Model Difference class with all differences into a JSON file. The CSV Summary Output Stage, however, creates a *.csv* file, which contains one row with the amount of missing and

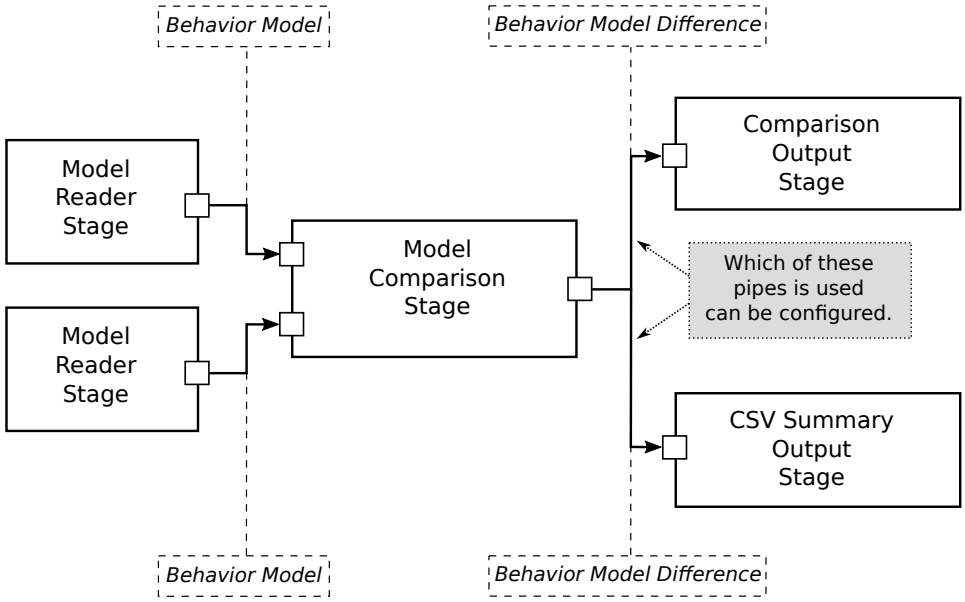


**Figure 4.2.** Overview of the TeeTime configuration of the Behavior Model comparison project.

additional nodes, edges, event groups and events. Additionally, the GED and the Jaccard-Index of the nodes and edges is written into the row. The Jaccard-Index between two sets $A$ and $B$ is defined as $J(A,B) = \frac{|A \cap B|}{|A \cup B|}$ [Jaccard 1902]. So, the Jaccard-Index of the nodes is the amount of shared nodes divided by the amount of total nodes and the Jaccard-Index of the edges is the amount of shared edges divided by the amount of total edges. Which of these sink stages is used can be set with command line arguments.

To make the detection of matching models easier, we created a shell script for pairwise comparison. For each of the IBMs, a *.csv* table is generated. Each row contains information about the similarity between the IBM and one of the cluster medoids. The iObserve comparison project with the CSV Summary Output Stage is used to generate these rows. The resulting tables were then analyzed manually to find the best match for each IBM.

## 4.2 Discussion of Experiment Results

The results of the evaluation are presented and discussed in this section. Section 4.2 is about the results of the fixed behavior workloads and Section 4.2 deals with the randomized behavior workloads.

**Table 4.1.** The clustering results for fixed behaviors. The used parameter are $\epsilon$=$\epsilon'$=9 and *MinPts*=7. The User Type column is the user type of the IBM, which is the best match for the cluster. The Ratio column shows how many of the expected behaviors are actually in the cluster. The GED and Jaccard-Index columns refer to the comparison results, between the cluster medoid and the IBM.

| Cluster | User Type | Ratio | GED to IBM | Jaccard-Index | |
|---|---|---|---|---|---|
| | | | | Node | Edge |
| Cluster 0 | Single Cat | 11/11 | 0 | 1 | 1 |
| Cluster 1 | New Customer | 11/11 | 13.1 | 1 | 1 |
| Cluster 2 | Account | 11/11 | 0 | 1 | 1 |
| Cluster 3 | Browser | 11/11 | 0 | 1 | 1 |
| Cluster 4 | Fish Lover | 10/11 | 16 | 1 | 1 |
| Cluster 5 | Cat Lover | 9/11 | 16 | 1 | 1 |
| Cluster 6 | Single Reptile | 11/11 | 0 | 1 | 1 |
| Noise | | 4/1 | | | |

## Fixed Behaviors

Table 4.1 shows the clustering results for the fixed behaviors. For each cluster the table shows the closest IBM, the amount of models in the cluster and the comparison results between the cluster medoid and the IBM. The comparison results include the GED and the node and edge Jaccard-Indices.

We chose an $\epsilon$ value of 9 and a *MinPts* value of 7. For the cluster extraction an $\epsilon'$ value of 9 was used. The $\epsilon$ value does not influence the clustering results, as long as it fulfills the constraint that it is larger or equal to the $\epsilon'$ value. When the $\epsilon'$ value is 10 or higher, the Single Reptile Buyer and Single Cat Buyer blend into one cluster, because the distance between the clusters is 10. When the *MinPts* value is higher than 10, the clusters can not be found anymore. The reason is that there are 11 behaviors of each user type, so there are no models with more than 10 neighbors. Below these values, the clusters are generated reliably, as the Behavior Models in a cluster are all very similar.

As all Jaccard-Indices are 1, all cluster medoids have the same nodes and edges as their corresponding IBM, which is a perfect result. The Single Cat Buyer, Single Reptile Buyer, Account Manager and Browsing User also have a GED of 0 to the corresponding cluster medoid, which means they are completely equivalent. However, the New Customer, Fish Lover and Cat Lover have higher distances. The difference between the New CustomerIBM and it's corresponding cluster medoid is that some events are stored in different edges. This was not further examined, so the cause can be as well in the manual IBM creation, as in the automatic workload generation. The problem with the Fish Lover and the Cat Lover is later discussed.

It can be seen that all 7 user-types were found by one of the clusters and no additional clusters were generated. As there were 11 behaviors of each user-type, the sizes of the clusters are also very good. Only the Cat Lover and Fish Lover miss 3 behaviors, which were

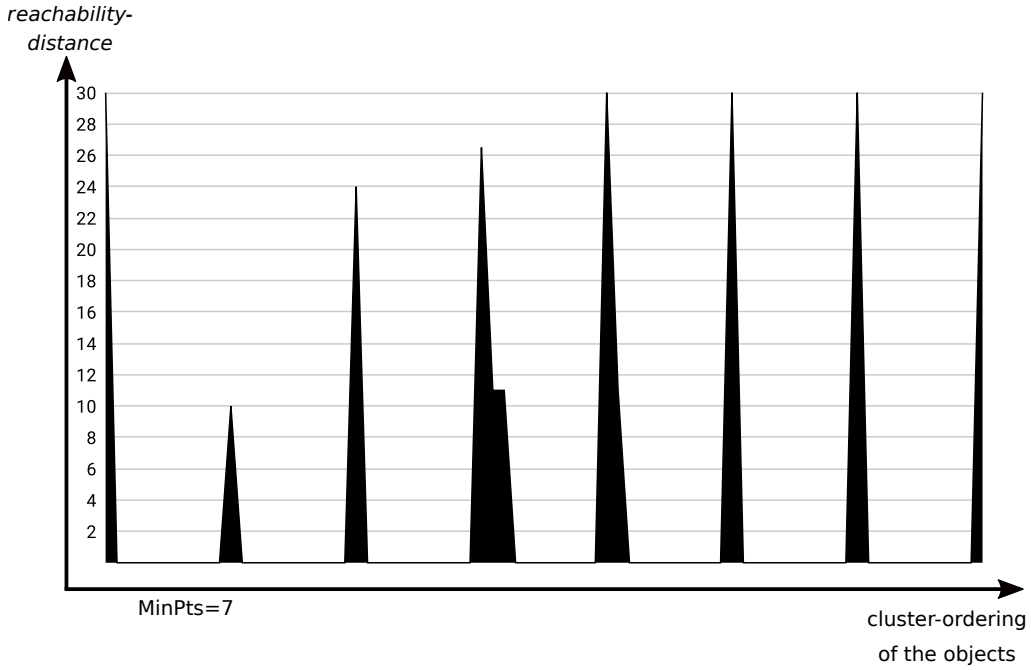


**Figure 4.3.** The OPTICS plot for the fixed behaviors. Each of the seven valleys correlates to one of the user types. We limited the *reachability-distance* to 30.

assigned to the noise. The cause for this problem is a confirmation event, after items are ordered. Only the three noise behaviors contain this event. This results in a distance of 11, so with an $\epsilon'$ value of 9 they are not considered as neighbors anymore. The confirmation event is still contained in the IBMs, because the event always occurred, when the behaviors for each user-type were generated separately.

The IBMs contain the confirmation event. This is the reason for the GED results of 11 in Table 4.1. We added the confirmation event to the IBMs, as the event always occurs, when the behaviors are generated separately. This indicates that the cause is the workload generation. It is still counter intuitive that the models are not placed in a cluster, because of one single event. Only with an $\epsilon'$ value higher than 11, they are added to the cluster, because the OPTICS algorithm will not consider them as neighbors otherwise. But, as mentioned before, from an $\epsilon'$ value of more than 10 on, the Single Cat and Single Reptile clusters merge into one.

One way to solve the problem in this specific case, is to adjust the GED costs. For example increasing the cost for a view category event would also increase the distance between the Single Cat and Single Reptile clusters, so the clusters do not merge anymore. But adjusting the metric does not avoid this kind of problem in general. As you can see

**Table 4.2.** Clustering results

| | 0 | 1 | 2 | 3 | 4 | 5 | 6 |
|---|---|---|---|---|---|---|---|
| **Account** | 263 | 286.1 | 0 | 261 | 274.5 | 274.5 | 263 |
| **Browser** | 310 | 338.2 | 261 | 0 | 326.7 | 326.7 | 316 |
| **Cat Lover** | 25.3 | 55.3 | 284.3 | 335.6 | 44.8 | 16 | 31.3 |
| **Fish Lover** | 31.3 | 55.3 | 284.3 | 335.6 | 16 | 44.8 | 31.3 |
| **New Customer** | 44 | 13.1 | 275 | 359.2 | 77.5 | 77.5 | 34 |
| **Single Cat** | 0 | 34 | 263 | 310 | 43.5 | 37.5 | 10 |
| **Single Reptile** | 10 | 24 | 263 | 316 | 43.5 | 43.5 | 0 |

in Figure 4.3, all clusters are entire in the OPTICS plot. So using a different approach to extract the clusters from the OPTICS plot would be a solution. For example the $\xi$-method described in Section 2.9 is good at detecting clusters with different density. This can be a good improvement in future work.

Although, there were only 77 behaviors execute in the workload generation, 78 Behavior Models showed up in the clustering results. The additional Behavior Model only contains events, which are created by the JPetStore initialization. As this does not represent an actual behavior, this should be filtered out in future work.

The distances between the clusters also convey a lot of information about the patterns in the data. For this purpose, we created a table with the pairwise distances between all IBMs and cluster medoids. It can be seen in Table 4.2. The pairs where the cluster matches the IBM are highlighted. One observation is that Single Reptile and the Single Cat are very close. This is expected, as the only difference is the parameters (e.g. CATS instead of REPTILES). Also the Single Cat is closer to the Cat Lover than to the Fish Lover, which shows that similar parameter lead to smaller distances. The highlighted cells are always the minimum in both, the rows and the columns. This means that by finding the best fitting cluster to an IBM, it results in the same pairs, as by finding the best fitting IBM to a cluster, so the matches are unique. This is not always the case. Another observation is that New Customer match (13.1), the Fish Lover match and the Cat Lover match (both 16) are larger than the distance between the Single Cat and the Single Reptile (10). This is counter intuitive, as matches should be near equal, if they are good. The reason is that, as laid out before, some mistakes can occur during both the workload generation and the manual IBMcreation.

## Randomized Behaviors

The *reachability-plot* of the OPTICS algorithm can be seen in Figure 4.4 . There is a valley for each of the 7 user types. The New Customer valley is smaller, as 11 instead of 41 of this user type were generated (cf. Section 4.1). The result of the cluster extraction can be seen in Table 4.3 . For each user type a cluster was generated. The Fish Lover misses 3 behaviors and the Account Manager misses 4 behaviors. Those behaviors are in the noise. Other than that,

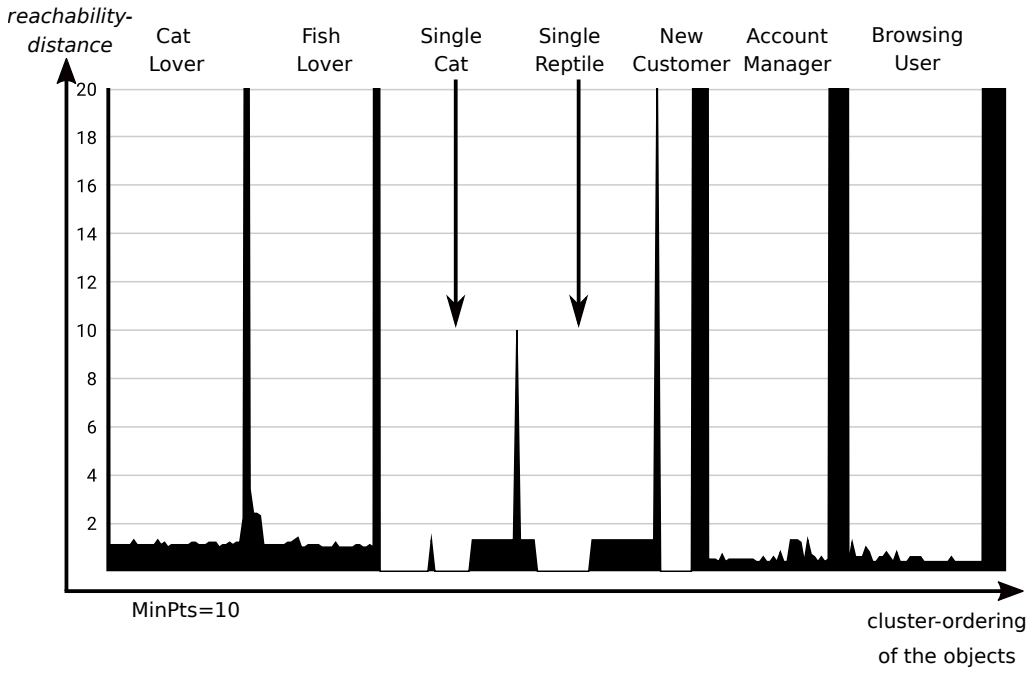


**Figure 4.4.** The *reachability-plot* for the randomized behaviors. There is a valley for each of the 7 user types.

all clusters are complete. There are 7 more Behavior Models in the noise. These correspond to the initialization of the JPetStore and not to user behaviors. There are 7 of these, as all user types were generated separately (cf. Section 4.1). Therefore, the JPetStore was initialized 7 times.

The OPTICS algorithm used an $\epsilon$ value of 9 and an *MinPts* value of 10. The clusters were extracted with an $\epsilon'$ value of 9. Similar results could be generated with different parameter values. The $\epsilon$ and $\epsilon'$ values are again equal, as a higher $\epsilon$ value does not influence the clustering results. All $\epsilon'$ values between 3.4 and 10 deliver the same results. A value below 3.4 leads to more noise objects, as some behaviors can not be assigned to the clusters anymore. Above a value of 10 the Single Reptile Buyer and the Single Cat Buyer merge into one cluster. Above a value of 21.3 the New Customer also merges with them. Above a value of 64.2 the Cat Lover and Fish Lover merge into one cluster. This is because those user types are considered as more similar according to the distance function. A *MinPts* value of 5 has no effect on the previous clusters. However, the initialization Behavior Models build a new cluster. This makes sense, as they are very similar. However, as these Behavior Models do not represent actual user behavior, they should be filtered out in future work. With a higher *MinPts* value than 10, the New Customer cluster is no longer generated. This is because there

**Table 4.3.** The clustering results for randomized behaviors. The used parameter are $\epsilon$=$\epsilon'$=9 and *MinPts*=10. The User Type column shows the user type, where the behaviors are from. The Ratio column shows how many of the expected behaviors are actually in the cluster. The GED and Jaccard-Index columns refer to the comparison results, between the cluster medoid and the IBM.

| Cluster | User Type | Ratio | GED to IBM | Jaccard-Index | |
|---|---|---|---|---|---|
| | | | | Node | Edge |
| Cluster 0 | Cat Lover | 41/41 | 244.2 | 0.6 | 0.44 |
| Cluster 1 | Fish Lover | 38/41 | 236.8 | 0.6 | 0.44 |
| Cluster 2 | Single Cat | 41/41 | 1.3 | 1 | 1 |
| Cluster 3 | Account | 37/41 | 29.2 | 1 | 0.83 |
| Cluster 4 | Browser | 41/41 | 2 | 1 | 1 |
| Cluster 5 | Single Reptile | 41/41 | 2.6 | 1 | 1 |
| Cluster 6 | New Customer | 11/11 | 13.1 | 1 | 1 |
| Noise | | 14/7 | | | |

are only 11 behaviors of that type, so no object has more than 10 neighbors. Apart from that, the clustering results do not change to a *MinPts* value of 30. Higher values were not evaluated.

A issue that can be seen is the similarity of the cluster medoids to the IBM in Table 4.3. Except for the New Customer, all distances are higher in comparison to the fixed behaviors in Table 4.1. The New Customer distance does not change, as it the only non randomized behavior. As the IBMs are the non randomized versions of the workload profiles, a higher distance is expected. However, the distances of the Fish Lover and the Cat Lover to their IBM are significantly higher. Therefore, the best matches for both IBMs are cluster 2. Cluster 2 is the behavior, which corresponds to the Single Cat Buyer. This shows that the current evaluation method alone is not sufficient to find the corresponding user types and clusters. Instead, we manually interpreted the clusters, where no suitable match could be found.

The reason for the large distances, is that the Cat Lover and the Fish Lover are supposed to log in and buy the products after the products are selected. Due to an issue of the workload generation, these actions were not executed. This issue was already fixed, however the behaviors were generated using the previous version. As the IBMs contain the actions, the Behavior Models are not considered as similar by the distance function.

## 4.3 Jira

Initially, we intended to analyze the Jira logs with our clustering approach. However, several obstacles hindered a thorough analysis.

a) The Jira logs do not contain session start and end markers. While the analysis stages are

able to synthesize these markers based on timeouts, there might be events missing or falsely assigned to a second Behavior Model.

b) The behavior of Jira regarding sessions is quite different to that of web shops. Thus, user sessions can last for days when the user never logs out. Therefore, sessions are not a good divider for behavior patterns. However, identifying other markers are not part of this thesis.

c) The data was made available at a rather late date within the thesis project.

Thus, we did not complete a thorough investigation of the behaviors.

Chapter 5

# Related Work

Vögele et al. 2018 proposed an approach called WESSBAS to automatically extract typical user behaviors from recorded session logs and simulate them for performance evaluations. The typical user behaviors are extracted by clustering the sessions using the X-Means algorithm. They represent the sessions in a matrix, which contains all transitions of a user session. These matrices are transformed into vectors and then compared using the euclidean distance metric.

This approach was later implemented in the iObserve. Dornieden 2017 extended the approach, by taking the parameters of the events into account. The evaluation showed that it is possible to detect predesigned users with the approach. However, the expected result could not be produced. Based on the WESSBAS approach, more clustering algorithms were implemented. The EM-Clustering performed worse than X-Means [Jung et al. 2017] and the results of the BIRCH algorithm were similar to X-Means [Lorenzen 2018]. They suspected that the result are limited due to the high amount of dimensions and the modeling of the behaviors. Therefore, Jannis 2018 proposed a new approach called Similarity Matching. The behaviors are represented in directed graphs and compared using a structural and a parameter distance function. The structural distance function compares the nodes and edges of the graph similar to the Jaccard-Index. The parameter distance function compares the parameters of the occurring events. Each cluster only contains behaviors, where both distances between all pairs of behaviors do not exceed a maximum value. The approach performed worse than X-Means. The concept of the approach is similar to our approach, as the behaviors are represented as graphs and the structure and parameters are compared. However, in contrast to our approach, the events were stored in the nodes instead of edges. Also, the clustering algorithm is very different.

There already exist similar approaches outside of the iObserve project. Hay et al. 2001 proposed to use the Sequence Alignment Method (SAM) to compare user behaviors. The SAM is similar to the GED, as sequences of events are compared by applying operations. The used operations were insertion, deletion and reordering of events. The difference of our approach is that behaviors are represented by graphs instead of sequences. Additionally, we evaluate the parameters and have a hierarchical structure. Through the hierarchical structure, events can have a similarity, instead of just being identical or different. They compared the SAM approach to an euclidean-based distance function and came to the conclusion it has a higher fitness for comparing behaviors.

A more recent approach uses a combination of three different proximity measures as

the distance function [Su and Chen 2015]. These measures are the frequencies of the events, the durations of the corresponding visits and common path length divided by the maximal path length. The purpose of the last measure is to take the event ordering into account. The behaviors are than clustered using a custom leader based algorithm. The evaluation showed that their approach could outperform previous clustering approaches – like k-medoids and the original leader algorithm – in effectiveness and efficiency.

Chapter 6

# Conclusions and Future Work

In this chapter we present the conclusions of our approach. The evaluation results are outlined in Section 6.1. In Section 6.2 our technical contributions to the iObserve project are described. Finally, in Section 6.3 we suggest future work, which might improve the analysis.

## 6.1 Summary of the Evaluation Results

The evaluation showed that the approach is able to detect clusters. In comparison to the previous approaches [Dornieden 2017; Jung et al. 2017; Lorenzen 2018; Jannis 2018], all predesigned user behaviors could be detected in fixed workloads. Additionally, we generated workloads, where the behaviors are slightly randomized. The approach was still able to detect all expected user types. However, some of the behaviors were considered as noise.

## 6.2 Technical Contribution

Our approach was implemented in the iObserve project using the TeeTime framework. We created stages for the Behavior Model generation, the M-Tree generation, the OPTICS algorithm, the extraction of clusters from the *reachability-plot* and the calculation of the medoid of the clusters. Additionally, we implemented a GED distance function. The parameters for the OPTICS algorithm and the GED can be configured using configuration files.

For the evaluation of the cluster analysis we implemented a comparison project in iObserve to compare two Behavior Models, which also uses the TeeTime framework. We than created a shell script for the pairwise comparison of Behavior Models, which can be used to find the best matching cluster for each user type. Additionally, we manually created an IBM for each user type.

## 6.3 Future Work

During the evaluation it was noticed that a Behavior Model is created by the initialization of the JPetStore. These events should be filtered out by the Session Acceptance Filter, as they do

not correspond to actual user behaviors.

Additionally, the Behavior Model can be improved. It can be relevant, how long a user visits certain servlets. Therefore, it would make sense, to add the durations of the visits to the nodes of the Behavior Model. Another possible extension is to add the information, in which node the User Session ends. This can be realized with an *End* node in addition to the *Init* node. Currently, an event group stores all the events that belong in it. However, it could be more elegant and efficient to only store different events. These events could than contain the information, how often it occurred.

The GED can also be improved. When two identical events have different predecessor nodes they belong to different edges. Currently, this means they are never compared. This could be changed by an operation that substitutes the source of an event. It could also make sense to weight the event groups differently. For example, an event that corresponds to the creation of a new account, is far more relevant, than than an event that corresponds to the visit of the main page. This could either be realized by a individually designed weight function for each application or by an approach that weights rare events more in general. In the Similarity Matching approach [Jannis 2018], differences in behaviors with a lot of actions have less influence on the distance. This could be a refinement for our approach, as large Behavior Models get more neighbors.

Currently, the only way to extract the clusters from the *reachability-plot* is the DBScan cluster extraction. However, the $\xi$-clusters extraction method is able to detect hierarchical clustering structures and clusters with varying densities. It could be evaluated, if it has a higher fitness at detecting user behaviors. Also, an output stage for the *reachability-plot* itself could simplify the manual cluster analysis.

The approach itself should also be further evaluated. The randomized workloads were generated by randomly varying the number of tasks performed by each behavior. Therefore, we have not evaluated, if our approach can cluster workloads with randomized parameters or randomized action orders. The approach should also be evaluated using larger and more realistic workloads, as we did not complete a thorough investigation of the Jira behaviors.

Finally, more distance measures and clustering algorithms could be implemented and evaluated. As the OPTICS algorithm works with any metric, further approaches can be combined with existing implementations.

Appendix A

# Configuration

The iObserve analysis project can be configured using Kieker configuration files. We added the following configuration keys with the prefix "org.iobserve.service.behavior.analysis." for our approach:

*epsilon* This represents the $\epsilon$ value for the optics clustering. It has to be greater than 0 and describes the maximal distance two neighbors can have.

*minPts* This represents the *minPts* value for the optics clustering. It must be a positive integer and describes the amount of neighbors a node must have, be be a core-node.

*maxModelAmount* This value determines, how many object can be clustered at the same time. If the number of objects is reached, the clustering can start. If this value is omitted, the clustering will only be started with the time trigger.

*outputUrl* The absolute path to the file, in which the results should be stored.

*nodeInsertionCost* The cost to insert an empty node to the behavior model. It should not be negative.

*edgeInsertionCost* The cost to insert an empty edge to the behavior model. It should not be negative.

*eventGroupInsertionCost* The cost to insert an empty event group to the behavior model. It should not be negative.

There are also configurations necessary for Kieker and the iObserve Behavior Analysis. These are the configurations we used:

*kieker.monitoring.metadata* Set to true.

*kieker.tools.source.LogsReaderCompositeStage.logDirectories* The absolute path to the log directories.

*iobserve.analysis.traces* Set to true.

*iobserve.analysis.dataFlow* Set to true.

*iobserve.analysis.singleEventMode* Set to true.

The stages we describe in Section 2.6 must also be configured. The configuration keys share the prefix "iobserve.analysis.behavior." and all values we use share the prefix "org.iobserve.analysis.systems.jpetstore.". Jannis 2018 further described the impact of the configuration values. These are the necessary configurations:

*IEntryCallTraceMatcher* The stage to transform the trance into events. For the JPetStore we use the value *JPetStoreCallTraceMatcher*.

*IEntryCallAcceptanceMatcher* This is used to configure the stage, which filters out unwanted sessions. We set it to *JPetStoreTraceAcceptanceMatcher*.

*ITraceSignatureCleanupRewriter* It is used to rewrite the operation names. We set it to *JPetStoreTraceSignatureCleanupRewriter*.

*IModelGenerationFilterFactory* This is used to define the events, which are filtered out. We set it to *JPetStoreEntryCallRulesFactory*.

*triggerInterval* The time in ms till the clustering should start.

**Appendix B**

# Workload Driver Bugs

- A slash was missing in most task URLs, so the tasks were not performed.
- The New Customer choose a password, which was too short. Therefore, no account was created.
- Each New Customer behavior tried to create an account with the same name. Therefore, only the first New Customer could create an account.
- When a behavior witnesses a read timeout, the project waits indefinitely long for the behavior to finish, so the project did not terminate.

## Appendix C

# Event Information

This appendix lists the expected events for each task. The events are ordered and the operations signature and parameter names are displayed.

**CreateNewCustomerTask**

| operations signature | parameter names |
|---|---|
| Catalog | |
| Account | signonForm |
| Account | newAccountForm |
| Account | username, password, repeatedPassword, account.firstName, account.lastName, account.email, account.phone, account.address1, account.address2, account.city, account.state, account.zip, account.country, account.languagePreference, account.favouriteCategoryId, newAccount, _sourcePage, __fp |

**LoginJPetStoreTask**

| operations signature | parameter names |
|---|---|
| Catalog | |
| Account | signonForm |
| Account | username, password, signon, _sourcePage, __fp |
| Catalog | |

**CheckoutJPetStoreTask**

| operations signature | parameter names |
|---|---|
| Catalog | |
| Cart | viewCart |
| Order | newOrderForm |
| Order | order.cardType, order.creditCard, order.expiryDate, order.billToFirstName, order.billToLastName, order.billAddress1, order.billAddress2, order.billCity, order.billState, order.billZip, order.billCountry, newOrder, _sourcePage, __fp |

**ChangeAccountInformationTask**

| operations signature | parameter names |
|---|---|
| Catalog | |
| Account | editAccountForm |
| Account | password, repeatedPassword, account.firstName, account.lastName, account.email, account.phone, account.address1, account.address2, account.city, account.state, account.zip, account.country, account.languagePreference, account.favouriteCategoryId, account.listOption, account.bannerOption, editAccount, _sourcePage, __fp |
| Account | editAccountForm |

**ViewOrderTask**

| operations signature | parameter names |
|---|---|
| Order | listOrders |

**ViewProductTask**

| operations signature | parameter names |
|---|---|
| Catalog | |
| Catalog | viewCategory, categoryId |
| Catalog | viewProduct, productId |
| Catalog | viewItem, itemId |

**AddItemToCartTask**

The last 3 events are repeated depending on the amount parameter.

| operations signature | parameter names |
|---|---|
| Catalog | |
| Catalog | viewCategory, categoryId |
| Catalog | viewProduct, productId |
| Cart | addItemToCart, workingItemId |

# Bibliography


[Ankerst et al. 1999] M. Ankerst, M. M. Breunig, H.-P. Kriegel, and J. Sander. Optics: ordering points to identify the clustering structure. In: *ACM Sigmod record*. Volume 28. 2. ACM. 1999, pages 49–60. (Cited on pages 10, 12, 13, 23)

[Assent 2012] I. Assent. Clustering high dimensional data. *Wiley Interdisciplinary Reviews: Data Mining and Knowledge Discovery* 2.4 (2012), pages 340–350.

[Becker et al. 2009] S. Becker, H. Koziolek, and R. Reussner. The palladio component model for model-driven performance prediction. *Journal of Systems and Software* 82.1 (2009), pages 3–22. (Cited on page 6)

[Ciaccia et al. 1997] P. Ciaccia, M. Patella, and P. Zezula. M-tree: an efficient access method for similarity search in metric spaces. In: *Proceedings of the 23rd International Conference on Very Large Data Bases*. VLDB '97. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1997, pages 426–435. (Cited on page 8)

[D'Avila 2013] E. R. D'Avila. *M-tree*. `https://github.com/erdavila/M-Tree`. 2013. (Cited on page 22)

[Dornieden 2017] C. Dornieden. Knowledge-driven user behavior model extraction for iobserve. Masterarbeit. Kiel University, June 2017. URL: `http://eprints.uni-kiel.de/38825/`. (Cited on pages 2, 5, 15, 25, 27, 35, 37)

[Ester et al. 1996] M. Ester, H.-P. Kriegel, J. Sander, X. Xu, et al. A density-based algorithm for discovering clusters in large spatial databases with noise. In: *Kdd*. Volume 96. 34. 1996, pages 226–231. (Cited on page 10)

[Hasselbring et al. 2013] W. Hasselbring, R. Heinrich, R. Jung, A. Metzger, K. Pohl, R. Reussner, and E. Schmieders. *iObserve: Integrated Observation and Modeling Techniques to Support Adaptation and Evolution of Software Systems*. Technical Report. Kiel, Germany: Christian-Albrechts-Universität Kiel, Oktober 2013. URL: `http://eprints.uni-kiel.de/22077/`. (Cited on pages 5, 6)

[Hay et al. 2001] B. Hay, G. Wets, and K. Vanhoof. Clustering navigation patterns on a website using a sequence alignment method. In: *Proceedings of 17th International Joint Conference on Artificial Intelligence*. 2001, pages 1–6. (Cited on page 35)

[Jaccard 1902] P. Jaccard. Lois de distribution florale dans la zone alpine. *Bull Soc Vaudoise Sci Nat* 38 (1902), page 72. (Cited on page 28)

[Jannis 2018] K. Jannis. Comparison of user behaviour classification methods. Bachelorarbeit. Kiel University, März 2018. URL: `http://eprints.uni-kiel.de/42869/`. (Cited on pages 1, 15, 20, 23, 35, 37, 38, 40)

[Jung and Adolf 2018] R. Jung and M. Adolf. The jpetstore suite: a concise experiment setup for research. *Symposium on Software Performance* (Nov. 2018). URL: http://eprints.uni-kiel.de/46706/. (Cited on page 4)

[Jung et al. 2017] R. Jung, M. Adolf, and C. Dornieden. Towards extracting realistic user behavior models. *Softwaretechnik-Trends* 37.3 (Nov. 2017), pages 11–13. URL: http://eprints.uni-kiel.de/40365/. (Cited on pages 1, 35, 37)

[Lorenzen 2018] M. Lorenzen. Classification of user behavior through connectivity-based clustering. Bachelorarbeit. Kiel University, Apr. 2018. URL: http://oceanrep.geomar.de/42870/. (Cited on pages 1, 15, 35, 37)

[Mahapatra and Chakraborty 2015] R. P. Mahapatra and P. S. Chakraborty. Comparative analysis of nearest neighbor query processing techniques. *Procedia Computer Science* 57 (2015), pages 1289–1298. (Cited on page 10)

[Newling and Fleuret 2016] J. Newling and F. Fleuret. A sub-quadratic exact medoid algorithm. *arXiv preprint arXiv:1605.06950* (2016). (Cited on page 24)

[Sanfeliu and Fu 1983] A. Sanfeliu and K. Fu. A distance measure between attributed relational graphs for pattern recognition. *IEEE Transactions on Systems, Man, and Cybernetics* SMC-13.3 (May 1983), pages 353–362. (Cited on pages 7, 19)

[*Selenium*]. *Selenium*. http://www.seleniumhq.org/. Accessed: 2019-08-24. (Cited on page 5)

[Shimizu et al. 2018] K. Shimizu, J. Landis, E. Macarron, R. Jung, I. AVE!, B. Raman, I. Baiborodine, Dongxu, Z. King, and S. Tripodi. Research-iobserve/jpetstore-6: single archive jpetstore with iobserve instrumentation (June 2018). (Cited on page 4)

[Struyf et al. 1997] A. Struyf, M. Hubert, P. Rousseeuw, et al. Clustering in an object-oriented environment. *Journal of Statistical Software* 1.4 (1997), pages 1–30. (Cited on page 24)

[Su and Chen 2015] Q. Su and L. Chen. A method for discovering clusters of e-commerce interest patterns using click-stream data. *electronic commerce research and applications* 14.1 (2015), pages 1–13. (Cited on page 36)

[Van Hoorn et al. 2012] A. van Hoorn, J. Waller, and W. Hasselbring. Kieker: a framework for application performance monitoring and dynamic software analysis. In: *Proceedings of the 3rd joint ACM/SPEC International Conference on Performance Engineering (ICPE 2012)*. ACM, Apr. 2012, pages 247–248. (Cited on page 4)

[Vögele et al. 2018] C. Vögele, A. van Hoorn, E. Schulz, W. Hasselbring, and H. Krcmar. Wessbas: extraction of probabilistic workload specifications for load testing and performance prediction—a model-driven approach for session-based application systems. *Software & Systems Modeling* (2018), pages 1–35. (Cited on pages 1, 35)

[Wulf et al. 2017] C. Wulf, W. Hasselbring, and J. Ohlemacher. Parallel and generic pipe-and-filter architectures with teetime. In: *ICSA Workshops*. IEEE Computer Society, 2017, pages 290–293. (Cited on page 3)