Comparison Of User Behaviour Classification Methods

Bachelor's Thesis

Jannis Kuckei

March 31, 2018

Kiel University
Department of Computer Science
Software Engineering Group

Advised by: Prof. Dr. Wilhelm Hasselbring
M.Sc. Marc Adolf
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, 31. März 2018

Abstract

User behaviour classification for software systems is a useful method that enables the understanding of a system's user base. The resulting classification of users into groups of similar behaviour can be used to generate realistic workloads for performance testing, or simply to aid in the improvement or addition of user-centric features. Many approaches to this problem use clustering algorithms which have trouble separating high-dimensional data. To address this challenge, we implement and evaluate an approach that compares and classifies user behavior models based on distance functions defined on those models.

Contents

Chapter 1

Introduction

1.1 Motivation

Knowledge of how users utilize software can aid in the diagnosis and prevention of problems. User behaviour classification groups users into classes of similar behaviour. Based on these classes, the user base of software systems as a whole can be understood better. For example, one can gain an understanding of how users utilize a certain feature, such as a search function. This can uncover strengths and weaknesses of said feature. The classification might show that most users have to use the search function repeatedly before they perform other actions, indicating that the search is not powerful enough. Furthermore, from groups of user behaviour models, workloads can be generated to make forecasts about the software system's performance. However, user behaviour is generally variable. It differs between individual users, and can also change over time. Therefore, monitoring and analysis are necessary to understand a user base and recognize changes.

Previous work on user behaviour classification has shown that clustering algorithms such as X-Means [Pelleg, Moore, et al. 2000; Moon 1996; Steinbach et al. 2004] have a problem with the high dimensionality of the input data modeling user behaviour. For X-Means clustering of user behaviour such as in the work of [Dornieden 2017], models describing the behaviour have to be converted into numerical vectors, which can grow very large. The vectors are then compared during clustering by metrics such as the gaussian distance metric. In this thesis, we implement a method we call Similarity Matching that aims to address the issue with large dimensionality other approaches have by directly comparing user behaviour models using distance functions defined on the models themselves, and grouping similar models together based on this distance.

We will implement our approach within iObserve [Hasselbring et al. 2013]. [Dornieden 2017] extended the iObserve framework's capabilities to collect additional user behaviour information. The call parameters of an operation called by a user can now also be taken into account when analyzing user behavior. This could be information such as which items a user purchased, that serve to further distinguish different behaviours. Our approach takes this information into account by defining distance functions both on the general structure of the user behavior, i.e., the operations called by the user, as well as the additional information pertaining to the operations themselves. This thesis has two goals:

1.2 Goal 1: Implementing Similarity Matching for User Aggregation in iObserve

We implement Similarity Matching and integrate it into iObserve using the TeeTime [*TeeTime Webpage*] framework.

1.3 Goal 2: Evaluation of the implemented Approach

In order to evaluate the relative success of this method to other approaches, we compare the results with those of [Dornieden 2017] using the JPetStore [*JPetStore GitHub Webpage*] application and similar workloads.

1.4 Document Structure

After this introduction, we provide the foundations of this thesis. The approach chapter describes the algorithm and its implementation. After that, we discuss the results in the evaluation chapter, and mention related work. We end the thesis with the conclusions, describing technical contributions and outlining future work.

Chapter 2

Foundations and Technologies

2.1 Behaviour Models

User behaviour models are abstractions to describe the behaviour of users or groups of users within a system. We will be using the following definition:

[Dornieden 2017] calls a behaviour model a directed graph $M = (N, E, \Delta, \Gamma)$, where N is the set of nodes and E the set of directed transitions. Each node represents an operation called by the user, and each edge $(s, t) \in E$ represents that the user called operation t directly after calling operation s at least once. The exact amount of transitions between nodes is described by the function Δ, which assigns each edge a natural number. Finally, the relation Γ is used to give each node, i.e. operation, one or more instances of call information (see Figure 2.1 for a graphical example).

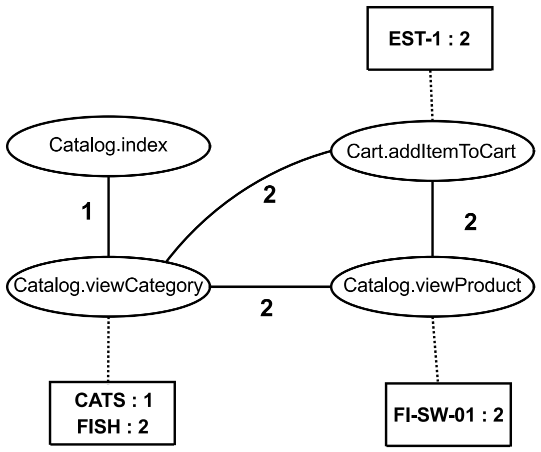

Figure 2.1. Example behaviour model of a JPetStore user

User behaviour can therefore be modeled as a directed graph with labeled edges. In this definition, the nodes (operations) also have labels, containing the call information. The motivation behind this is that user behaviour is described with more detail. For example, if one such node models the operation of purchasing an item, then the call information could model what item has been bought by the user, allowing for a finer distinction of interest groups among users (e.g. people who tend to buy specific items every time they shop). The result of our algorithm is a collection of these models. Furthermore, we require a number of models as input, which are generated from monitored user sessions.

2.2 TeeTime Framework

TeeTime [*TeeTime Webpage*] is a framework available for Java that provides an implementation for the Pipes-and-Filters architecture. This pattern is commonly used to process large amounts of data. TeeTime was developed after the previous implementation of the Pipes-and-Filters architecture for Kieker [van Hoorn et al. 2012] proved to be inadequate for live architecture-recovery and other complex tasks. [Wulf et al. 2014]

The basic idea of this pattern is a series of filters connected by pipes, where the pipes represent the flow of data between the filters. The filters process data from one or more pipes, and send it to one or more output pipes (see Figure 2.3). In TeeTime, filters are called stages. Stages can have multiple ports, which act as their inputs and outputs. Pipes connect to these ports, allowing each stage to have multiple pipes going in and out. [Wulf et al. 2014]

TeeTime offers concurrent execution of stages with different approaches as to how sending elements through pipes is handled. No dedicated scheduler is used for this in order to reduce complexity and increase throughput. It also includes a type-safety system guaranteeing that no incompatible ports will be connected through pipes to reduce debugging needs. [Wulf et al. 2014]

As a further abstraction, TeeTime directly supports stage composition. Since a system of stages and pipes is characterized from an outside perspective merely by its input and output ports, it can be considered a stage and used as such. This hierarchical approach fits neatly into the pattern as depicted here. [Wulf et al. 2014]

iObserve uses TeeTime to structure the process of analysing the monitoring data and generating models at runtime [Wulf et al. 2014]. As an example, in their implementation of user behavour aggregation, [Dornieden 2017] creates a composite preprocessing stage that takes as an input objects resultant from the monitoring. It consists of four smaller stages and serves to prepare the user data for aggregation in another stage after it. We use TeeTime to implement and connect our approach to iObserve in this thesis.

2.3 iObserve Research Project

iObserve [*iObserve Webpage*] is a research project dedicated to developing new technologies to support the adaptation and evolution of long-living software systems. The motivation behind this project is the expectation that future services will largely run on the cloud, or in other words, third party infrastructure. This poses a challenge to the observation and analysis of such systems, and in return the modification to address potential issues. An example of such an issue that operators of a software system would wish to prevent is the violation of data geo-location policies. When under heavy load, a cloud-service provider may attempt to clone a set of data to make it available through other servers. Laws such as the *Bundesdatenschutzgesetz* make it illegal to move confidential information to servers outside of European countries that follow certain standards regarding privacy regulations. [Hasselbring et al. 2013]

iObserve follows a MAPE-K approach [Kephart and Chess 2003], combining monitoring, analysis, planning, execution and (domain-specific) knowledge. For monitoring, Kieker [van Hoorn et al. 2012] is used to instrument the software to be observed. To make a common approach possible to a task highly dependent on the software that is to be monitored, two languages have been developed - the Instrumentation Record Language for notation of monitoring data, and the Instrumentation Aspect Language which describes the application of probes to a software model. [Hasselbring et al. 2013]

Analysis takes into account the aforementioned data location policies, and matters such as service-level agreements regarding cost and performance. A way in which iObserve does its analysis is to generate and update models (at runtime) for decision-making. It also uses Palladio to create and analyse design phase models which help with performance and maintainability predictions. [Hasselbring et al. 2013]

Planning and execution refer to the ability of iObserve to automatically adapt to changes based on the results of the analysis stage [Hasselbring et al. 2013]. This can refer to measures taken to prevent the violation of data geo-location policies.

Alongside automatic changes, part of the iObserve life-cycle seen in figure Figure 2.2 is the evolution of the software system. Evolution describes the evaluation of the analysis by an operator or developer alongside with the manual realization of changes deemed necessary by them [Hasselbring et al. 2013]. Unlike adaptation, this part requires human intervention. Using an example involving user behaviour: An operator noticing a shift in the most common user group of an online shop - from users buying single items and logging out to users browsing the inventory - may decide to add functionality suggesting item purchases to users to entice them to spend more money.

The analysis in iObserve is structured in the form of a pipe-and-filter architecture using the TeeTime framework. The monitoring data to be analyzed is received through a source stage implementing the *ISourceCompositeStage* interface. This requires the source stage to provide an output port of the type *IMonitoringRecord*, which is the common interface for a variety of monitoring data. Two possible configurations for the source stage are a stage reading from a network connection, or a stage that reads the monitoring data from a file system. The monitoring records are then fed into the *RecordSwitch* stage, where they are sent to different output ports based on their type. For analysis of user behaviour, the two types of monitoring records we are concerned with are *ISessionEvent* and *IFlowRecord*. Session events signify the start and end of user sessions, and flow records contain the information of user activity. The objects implementing *IFlowRecord* are further processed by the *TraceReconstructionCompositeStage*, and we connect our approach to these stages as

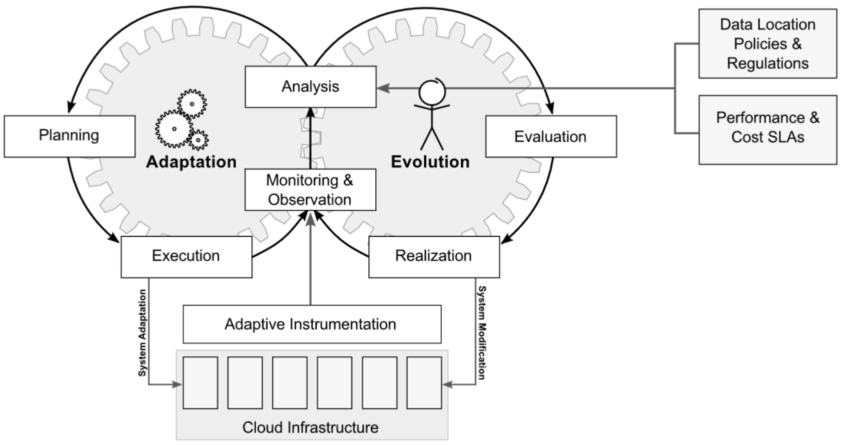

Figure 2.2. The iObserve life-cycle [Hasselbring et al. 2013]

the composite *BehaviorCompositeStage*. Figure 2.3 depicts the stages that are relevant to the thesis.

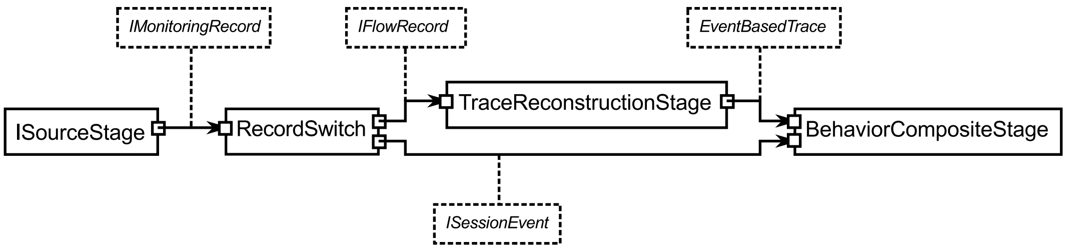

Figure 2.3. Stage diagram of relevant portion of iObserve analysis stages. Note that *RecordSwitch* has more than two output ports for other monitoring records, which are not depicted.

2.4 JPetStore

JPetStore [*JPetStore GitHub Webpage*] is a simple web application that implements a pet store. Users can browse products, add them to their cart, register an account, log in, place orders, and so on. It has been instrumented for iObserve with Kieker [van Hoorn et al. 2012] so that user actions can be monitored. We do not use JPetStore directly, but have access to monitoring data captured in the application that we use to test our method. Some operations in JPetStore contain additional call information. An example of this is the operation of viewing a category, which supplies the viewed category [Dornieden 2017]. We discriminate user behaviour based on this additional information, as well.

Chapter 3

Behaviour Clustering Approach

In this section, we will describe the algorithm we call *Similarity Matching*. The idea behind this algorithm is to group together behaviour models based on their distance to each other as defined in two metrics, and to then generate behaviour models for those groups to describe their shared behaviour. First, we give an informal description of how it functions to allow the reader to form a basic understanding. The section after that describes the algorithm more formally.

3.1 Underlying Idea Of The Algorithm

The purpose of Similarity Matching is to classify a set of user behaviour into groups based on similarity between behaviours, and to then generate a representation of behaviour for each of those groups. Therefore, the necessary input is a collection of behaviour models, and the output is an arrangement of these models into groups, as well as a representative behaviour model for each group. A measure of similarity between models is provided by two functions that we call *structural distance function* and *parameter distance function*. The *structural distance function* calculates a distance between two behavior models based on the general structure, that is, the nodes, edges and edge numbers of the directed graphs. The distance between call parameters of two behaviour model is computed by the *parameter distance function*. The *parameter distance function* must be implemented specific to the observed software system, as the semantics of the parameters and relationships between their values affect the distance. We provide an implementation for JPetStore [*JPetStore GitHub Webpage*], while implementing the other distance function as application-agnostic.

Our Similarity Matching approach comprises of three stages which we call Vectorisation, Matching and Model Generation respectively. In Vectorisation, a vector for each behaviour model is generated which contains the distance to each other model according to both distance functions. During Matching, these vectors are compared component-wise and vectors with a distance that does not exceed a pre-set maximum value (per metric) are grouped together. Finally, for every group, a behaviour model is generated.

3.2 Formal Description

The Similarity Matching algorithm expects a set of user behavior models M (defined as in Section 2.1) and two real numbers r_p, the parameter radius, and r_s, the structure radius, as input. The radii are used during Matching to determine how far apart two vectors can be to still be considered similar.

3.2.1 Vectorisation

Let $n = |M|$ be the amount of models this algorithm receives as input. Furthermore, let $d : M \times M \mapsto \mathbb{R}$ and $p : M \times M \mapsto \mathbb{R}$ be the *structural distance metric* and *parameter distance metric* respectively. We assume an ordering of the elements of M so that each element $m^i \in M$ can be identified by a number $i \in \{1, ..., n\}$. During this stage, we assign each model $m^i \in M$ a vector $v^i \in \mathbb{R}^{2n}$, such that $v^i_{2k-1} = d(m^i, m^k)$ and $v^i_{2k} = p(m^i, m^k)$. In words, the vector v^i represents the distances to other models for the model m^i. This also means that $v^i_{2i-1} = v^i_{2i} = 0$ since a model is completely similar to itself. Please note that the subscript of the vector indicates the component, whereas the superscript refers to the model it belongs to. The result of this stage is a set of vectors $V = \{v^1, ..., v^n\}$ representing every behavior models distance to each other.

3.2.2 Matching

In this stage, we assign each model m^i to a group based on its vector v^i. Let the initial set of groups be $\mathcal{G} = \{\{m^1\}\}$, that is, containing a single group with the first model. We iterate over the rest of the models in order, looking for a group $G \in \mathcal{G}$ such that for our current model m^i, the following conditions holds true:

1. $\forall m' \in G : \forall k \in \{0, 2, ..., 2n-2\} : |v^i_k - v'_k| \leqslant r_s$

2. $\forall m' \in G : \forall k \in \{1, 3, ..., 2n-1\} : |v^i_k - v'_k| \leqslant r_p$

where v' is the vector belonging to m'. This means that the absolute distance between every vector component must lie below the specified radius for each metric. Since there can be multiple matches, for the sake of consistent results, we choose the first one. If no group matches the criteria, a new group containing m^i is created. The process is then repeated until all models have been assigned to a group.

3.2.3 Model Generation

Lastly, after grouping together similar behaviours, we generate a single behaviour model that describes the group itself. For this, we introduce two approaches we call union- and intersection method respectively, which are named after the corresponding set operations.

Let the group G be a non-empty set of models for which we wish to generate a group model $M = (N, E, \Delta, \Gamma)$.

Union method This method defines the group model so that for every $m \in G$ with $m = (n, e, \delta, \gamma)$, the following conditions hold:

1. $n \subseteq N$
2. $e \subseteq E$
3. $\forall k \in e : k \in E \implies \delta(e) \leqslant \Delta(e)$
4. $\gamma \subseteq \Gamma$

This means that every model of the group is contained in the group model in the sense that you could remove nodes, edges, call-information and edge-numbers from the group model and end up with a model from the group it represents. We call this method the union method because it combines all models of the group.

Intersection method Here, only the mutual features are combined into the group model. The intersection method defines the group model so that for every $m \in G$ with $m = (n, e, \delta, \gamma)$, the following conditions hold:

1. $N \subseteq n$
2. $E \subseteq e$
3. $\forall k \in E : k \in e \implies \Delta(e) \leqslant \delta(e)$
4. $\Gamma \subseteq \gamma$

Instead of every model being contained in the group model, as is the case with the union method, the group model is generated so that it is contained in every model it represents. The group model is the intersection of features between all models. Therefore, every model of the group can be conceived by adding more features to the group model generated by the intersection method.

3.3 Algorithm Design and Implementation

This section describes the implementation of the algorithm and necessary prerequisites. We begin by talking about the behaviour model implementation and the distance functions. The stages of the algorithm are defined and finally, we elaborate on how to configure the analysis.

3.3.1 Behavior Model Implementation

The existing iObserve implementation of user behaviour models consists of a collection of nodes and edges as a set [*iObserve Analysis Repository*]. The nodes contain an operation signature and a set of their call information, while the edges contain a source and destination node, as well as an edge number to denote how often the edge is traversed. Unfortunately, there are shortcomings with how the call information is stored and added to existing nodes. We duplicate the model classes into a different package because existing algorithms rely on them being unchanged and rewrite some of their functionality to suit our approach. How our new classes are different and the reason for those changes is addressed in the following subsections.

BehaviorModel

This class represents a single behaviour model. It stores *EntryCallNode*- and *EntryCallEdge*-objects, representing nodes, edges and call information of a behaviour model, and provides methods for adding and accessing them.

The previous implementation stores these nodes and edges in collections of the *HashSet* type. In Java, *HashSet* implements the *Set* interface. This has the effect that any nodes added to an existing behaviour model that already contains a node with the same operation signature will not be added. The preferred behavior is to combine them by merging their call information. Similarly, adding additional edges between the same operations should increase the edge numbers. However, when adding a node or edge, the implementation performs a search on the whole set looking for a node or edge with similar properties in order to see whether the new element would need to be merged. This is, at best, a linear operation over the size of the set in question and means that using a *HashSet* offers no real advantage.

We store the nodes and edges in collections of the type *HashMap*. Here, we map between the type *String* and *EntryCallNode* or *EntryCallEdge* respectively. The collection offers (in practice) constant access time to elements [*HashMap Javadocs*]. The key used to map to nodes is their operation signature. This means that when adding a new node, it is first checked whether the signature of that node already maps to a value. If it does, those nodes can be merged. If there is no corresponding value, the new node is added. Similarly for edges, the key used to store them are the signature of the source node and the signature of the edge node, appended to each other in that order. We add an additional parameter of type *boolean* to the methods used to add nodes and edges that decides whether adding already existing nodes should have their call information merged, or be ignored.

The operation of adding a node or edge that already exists now happens in constant time, as opposed to linear time. Since *HashSet* internally uses a *HashMap*, the operation of adding a new node or edge should practically be the same as the previous implementation [*HashSet Javadocs*].

The previous implementation also has a method to find a node in the model. We change the parameter of this method to accept an operation signature of type *String* instead, and also add a method to find an edge based on the signatures of its source and target nodes. These methods also gain a significant speed advantage based on the new data structure used, for the same reasons as explained above.

EntryCallNode

This class represents operations, storing the operation signature and any call information it has been called with. The signature is stored as a *String* and the info as a collection of *CallInformation*.

The previous implementation does not allow for more than one type of call information to be stored in a single node. This would, to use a JPetStore [*JPetStore GitHub Webpage*] example, prevent the model from describing a situation where a user buys two different items. The *CallInformation*-objects are stored in a *HashSet*. When adding new call information to a node, information of a signature that already exists would be ignored by the method responsible for this task.

Just like with the collections for *BehaviorModel*, we replace the collection with a *HashMap* for the same reasons (see Section 3.3.1). We also remove the restrictions for only being able to add call information with a new signature and add a method to search for call information based on its signature and value. The key for accessing call information in the map is a *String* of the call information's signature and parameters, appended to each other in that order.

CallInformation

This class represents the call information passed alongside an operation call by the user. It consists of a parameter type called the signature, as well as the actual parameter value. For example, a user performing a purchase action may pass along call information with the type *itemId* and the value *EST-1*.

The previous implementation stores the signature as a *String* while the value is stored as a *double*. Storing the value as a numeric type is necessary for the X-means based approach [Dornieden 2017] used. We store both the signature and value as *String*, like they are received from the monitoring. A numerical value is not needed and creates unnecessary dependencies on the encoding used when implementing the distance functions for behavior models. We also add a counter indicating the repetition of the call information (type *int*), and have added methods for the manipulation of this value.

EntryCallEdge

This class represents an operation call, initiated from a previous operation. It models the edges and edge numbers of a behavior model and contains two nodes (source and destination) of type *EntryCallNode* as well as a number of type *int* for counting how often the edge has been traversed, i.e., one operation called from another.

We changed the type of the edge number from *double* to *int*, because there is no interpretation for traversing an edge a fraction of times in accordance with the behavior model we used.

3.3.2 Distance functions

We implement an example for both the *structural distance function* and *parameter distance function*. Our *structural distance function* does not make any assumptions about the user behaviour models. The *parameter distance function* is implemented specifically for JPetStore *jpetstorepage*.

Structural distance function

Let A and B be the behaviour models this function receives as input, with N_A and N_B being the amount of nodes for each model, and S being the amount of nodes they share. The distance function first calculates a node distance as such: $N_{distance} = \frac{N_A+N_B-S}{N_A+N_B}$

This yields a value between 0 and 1, for when the nodes are the same to when they are completely non-intersecting. After this, we calculate an edge distance in exactly the same manner, counting edge numbers so that an edge number of n means that the edge exists n times. The final distance return by this function is the average of the node and edge distance. As the maximum of the edge distance is 1 as well, the maximum value of the total distance has the same value.

Parameter distance function

The call information for JPetStore operations describes one of three things:

Category This distinguishes items at a species level. Examples include *FISH, CATS*.

Product This is a finer distinction between items in the same category - for example, fresh-water and salt-water fish.

Item This is the finest distinction between animals sold by the store, belonging to both a specific product and category.

For a single operation, the call information is homogeneous by type, i.e., an operation will not include mixed types such as a category and a product. Therefore, we first define a distance function between call information, defining it differently for every type. For all types, we define the distance between call information to be 0 if they are equal to one another. If they are not equal, they are defined as such: For categories, we define the distance to be 1 if the categories are different. For products, we define the distance to be 1 if the products are in the same category, and 2 if they are in different categories. For items, we define the distance to be 1 if they are within the same product, to be 2 if they are within different products but the same category, and the be 3 if they are within different categories.

To compare two models, we compare every node with the same operation signature. Should two models not share a node, we compare it with a dummy node containing no call information. If we omitted the comparison for nodes that are not shared, then the distance between models with a lot of call information and little call information on an operation would be larger than the distance between models with a lot of call information and a model that does not contain the operation at all. This could, for example, lead to the undesired effect of models where a user buys one item and models where the user buys ten items having more distance than the distance between models where the user buys ten items and a model where the user buys no items at all.

When comparing two nodes, we find the permutation of call information in the respective nodes so that the sum of distances between call information is minimal. We do this by first finding a match for all the call information contained in the other node, where the respective distance would be zero, and then continuing to find matches to keep the respective distance as minimal as possible. We do this until all call information on at least one node has been matched. Because the nodes can contain different amounts of call information, this leaves the possibility that we have a remaining set of call information that has not been matched. We add a distance of 4 for every instance of unmatched call information. After comparing every node, we sum the distances of each comparison and divide it by the total amount of node comparisons. This is the final distance value calculated by this method.

3.3.3 Converting user sessions to behavior models

So that the algorithm can work with user sessions, they first need to be converted into behaviour models that represent the behaviour of that single user during their session. For this, we supply a class *UserSessionToBehaviorModelConverter* that provides a static method taking a single *UserSession*-object and and returning a *BehaviorModel*-object. This is done by iterating over the user session's list of events (type *PayloadAwareEntryCallEvent*), which are in chronological order, and creating *EntryCallNode*-objects from them. Those nodes are then put into *EntryCallEdge*-objects in the order of how their corresponding events were stored, and added to an initially empty behavior model. Any issues with duplicate nodes or edges are automatically handled by the *BehaviorModel* classes' methods for adding edges (see Section 3.3.1).

3.3.4 Stages

This section describes the stages which implement Similarity Matching. In Section 2.3, the relevant portion of the top-level stage design for the analysis was shown. In it, we integrate our approach as the *BehaviorCompositeStage*. This stage consists of three sub-stages, which can be seen in Figure 3.1. The first stage pre-processes the monitoring records into user sessions, which list the user actions in a chronological manner. The middle stage converts these into behaviour models and performs our approach for classification. This stage implements the *IClassificationStage* interface and provides input ports for a timestamp and user sessions, as well as an output port for behaviour model arrays. The last stage then writes the results into files. Our thesis was written in parallel to another thesis working on user behaviour classification, and we decided to use this common interface in order to keep the stage design tidy and avoid duplicate code.

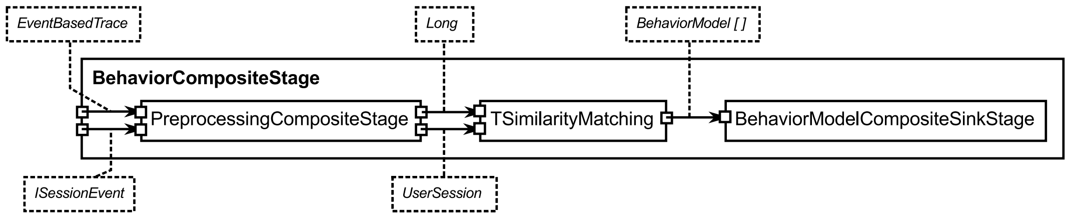

Figure 3.1. Stage diagram of *PreprocessingCompositeStage*

PreprocessingCompositeStage

Before any user behaviour classification can be done, the preprocessing stage transforms the relevant monitoring records and traces into user sessions. These represent the chronological actions of a single user during a unit called a session. A session is completed either when it ends through an action by the user or system (e.g. logging out) or when inaction by the user terminates the session. Most of these stages are already part of iObserve [*iObserve Analysis Repository*], but some of them had to be modified to work with our approach.

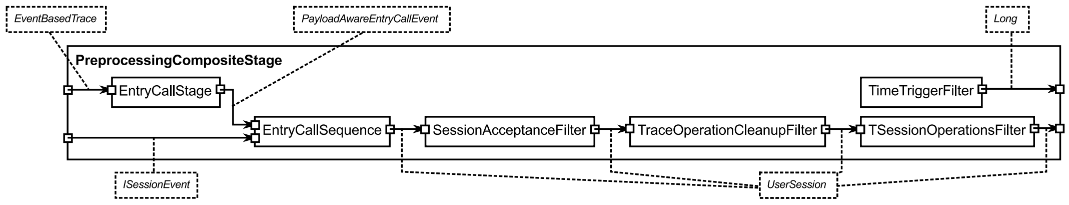

Figure 3.2. Stage diagram of *PreprocessingCompositeStage*

EntryCallStage This stage takes objects of type *EventBasedTrace* generated by the stage *TraceReconstructionCompositeStage* and create objects of type *PayloadAwareEntryCallEvent*. These events can represent direct user actions, or passive events such as the loading of resources (e.g. an image). The structure includes the operation signature and call parameters characterizing the event as well as the user they belong to.

EntryCallSequenceFilter This stage accepts the entry call events from the previous stages as well as objects of type *ISessionEvent* from the *RecordSwitch*. It aggregates them into a *UserSession*, which is a chronological collection of the events of a particular user. They are kept until either a *SessionEndEvent* is received, or the session expires. Then, they are sent to the next stage. Because we are using previously recorded monitoring data in our evaluation, we temporarily remove the functionality of this stage to remove expired sessions, as the data we use contains timestamps that would mark it as expired. This would otherwise cause the data to be ignored by the stage.

SessionAcceptanceFilter This stage uses a configurable object of type *IEntryCallAcceptanceMatcher* to remove user sessions containing unwanted events from the analysis entirely. We include this stage for future use, but configure it with a *JPetStoreTraceAcceptanceMatcher* to let every session pass through. Instead, we later filter unwanted events in the stage *TSessionOperationsFilter* without dropping sessions.

TraceOperationCleanupFilter Just as with *SessionAcceptanceFilter*, we merely include this for future use and configure it with a *JPetStoreTraceSignatureCleanupRewriter* so that it will have no effect. This stage can be used to rewrite the properties operation/class signature of entry call events in user sessions. This is not necessary for our implementation, but was included in the configuration of another user behaviour classification implementation in iObserve upon which we based our setup [Dornieden 2017].

TSessionOperationCleanupFilter For user behaviour classification, we are only interested in events that describe a direct action of the user. This stage removes unwanted events from user sessions using a configurable object of type *EntryCallFilterRules* based on their operation signature. We supply this stage with filter rules generated from *JPetStoreEntryCallRulesFactory*. That way, events that describe the loading of images and stylesheet resources are removed, as they do not represent a direct user action, but merely a consequence of it.

TimeTriggerFilter This stage periodically sends a signal (timestamp) to its output port. This is meant to trigger the classification stage during live deployment of the analysis, so that it can collect user behavior models for a certain duration, and only periodically initiate the classification process. For purposes of the evaluation, this stage currently only sends its signal once. As it is an active producer stage, repeated production of a signal would otherwise cause the analysis not to terminate, which is a problem for the evaluation, which requires the analysis to be started many times.

TSimilarityMatching

After preprocessing of the monitoring data, we now have a stream of user sessions that we can work with. The design of the composite stage for the actual processing follows almost directly from the partition of the algorithm as described in Section 3.2. However, since the input is of type *UserSession*, a stage to convert these sessions into behaviour models is required.

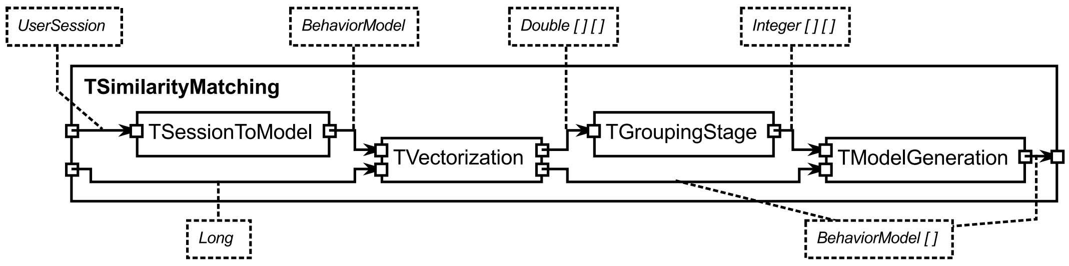

Figure 3.3. Stage diagram of *TSimilarityMatching*

TSessionToModel This stage takes user sessions and converts them to behaviour models, using the *UserSessionToBehaviorModel* class (see Section 3.3.3). Any incoming objects of class *UserSession* are immediately converted to a *BehaviorModel* and sent to its output port.

TVectorization This stage is configurable with two objects implementing the *IParameter-MetricStrategy* and *IStructureMetricStrategy* respectively, and implements the vectorization described in Section 3.2.1. The objects provide a method for comparing two behaviour models and returning a distance value of type **double**. The models are collected by order of arrival in this stage. We use a simple *ArrayList* for this. Any time a new model arrives, a vector (as a list) is created and filled with the structure/parameter distance of the new model relative to each previously collected model, in the order of their arrival. This ensures that each position in the vector corresponds to the same model. Finally, the last two vector positions are filled with the value 0, signifying the model's distance to itself. Eventually, the timer stage will send a signal that it is time to process all of the collected models and vectors. Upon that signal, this stage will convert all of the collected models into an array and send them to the *TModelGeneration* stage. The generated vectors are converted to arrays as well and sent to the *TGroupingStage*. In order to prepare for the next batch of models, the internal collection is reset to an empty list.

TGroupingStage This stage is configurable with two values of type **double** which specifiy the radii parameter as discussed in Section 3.2. Here, every vector is assigned to a group based on its similarity to other vectors as described in Section 3.2.2. The groups are stored as an *ArrayList*, where every group is again an *ArrayList* storing type *Integer*. The values in those groups are the indexes of the vectors in the collection received by this stage. The vectors are processed in order, and a subroutine is called that attempts to find a group. If no group is found, a new one is created. Once all vectors have been assigned a group, the group list and contained groups are converted into an array and sent to the *TModelGeneration* stage.

TModelGeneration This stage implements the model generation after the initial models have been grouped together as seen in Section 3.2.3. It is configurable with an object implementing the *IModelGenerationStrategy* that provides a method taking an array of behaviour models and returning a single behaviour model, the representative model. On receiving both the array of behavior models, as well as the array of groups, the stage generates a behavior model array in accordance with the group assignment made by the previous stage. Since the vectors that had been grouped were generated in the same order as the behavior models had been collected, the indexes corresponding to the vectors directly identify the models belonging to each vector as well. Thus, the stage only has to iterate over every group and pick the models in the model array according to the stored indexes. These are then fed into the model generation method, and the resulting models are sent to the output port.

BehaviorModelCompositeSinkStage

This stage (see Figure 3.4) takes an array of behaviour models and writes them to the file system the analysis is running on. It is configurable with an output folder path. The resulting model files follow the scheme "model_*.txt", numbered starting at 0. In order to be able to use the *BehaviorModelSink* stage, which serializes and saves the models as JSON, writing and accepts single behaviour models, we added an additional stage *BehaviorModelDecollectorStage* that takes the array of models generated by the classification stage and sends the elements to its output port individually.

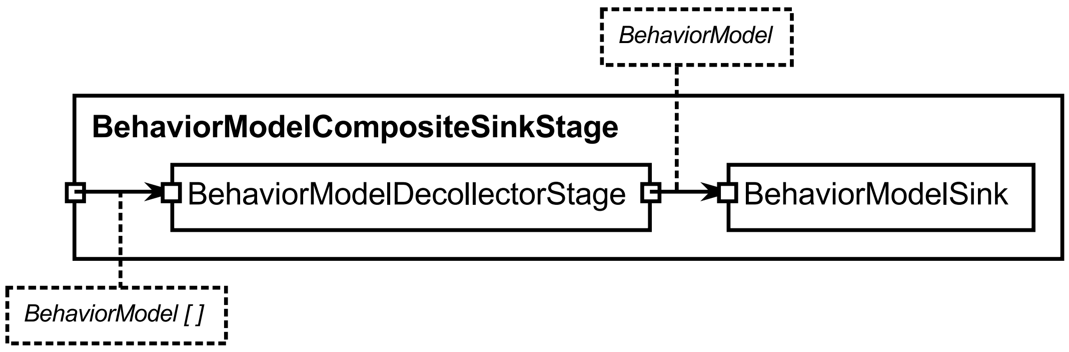
BehaviorModel
BehaviorModelCompositeSinkStage
BehaviorModelDecollectorStage
BehaviorModelSink
BehaviorModel [ ]

Figure 3.4. Stage diagram of *BehaviorModelCompositeSinkStage*

3.3.5 Configuration

The following are all the configurable parameters we added for our implementation and their default settings for the evaluation. All of these configuration keys share the prefix *"iobserve.analysis.behavior."*.

filter
The name of a class implementing *IBehaviorCompositeStage*. This class is the top-level stage containing the whole of the behaviour classification, including pre-processing and the output stage.

Value: *org.iobserve.analysis.clustering.filter.similaritymatching.BehaviorCompositeStage*

IEntryCallTraceMatcher
The name of a class implementing *IEntryCallTraceMatcher*. This is used by the **EntryCall-Stage** to identify entry call events.

Value: *org.iobserve.analysis.systems.jpetstore.JPetStoreCallTraceMatcher*

IEntryCallAcceptanceMatcher
The name of a class implementing *IEntryCallAcceptanceMatcher*. This is used by the *SessionAcceptanceFilter* stage to decide which sessions to remove from analysis based on the operation signature of their events.

Value: *org.iobserve.analysis.systems.jpetstore.JPetStoreTraceAcceptanceMatcher*

ITraceSignatureCleanupRewriter
The name of a class implementing *ITraceSignatureCleanupRewriter*. This is used by the *TraceOperationCleanupFilter* stage to rewrite certain event properties.

Value: *org.iobserve.analysis.systems.jpetstore.JPetStoreTraceSignatureCleanupRewriter*

IModelGenerationFilterFactory
The name of a class implementing *IModelGenerationFilterFactory*. This is used by the *TSessionOperationsFilter* to create filter rules that decide which events to remove from sessions and which to keep.

Value: *org.iobserve.analysis.clustering.filter.models.configuration.examples.JPetStoreEntryCallRulesFactory*

sm.IParameterMetricStrategy
The name of a class implementing *IParameterMetricStrategy*. This is used by the *TVector-izationStage* stage to determine the distance between behaviour models based on their event's parameters.

Value: *org.iobserve.analysis.clustering.filter.similaritymatching.JPetStoreParameterMetric*

sm.IStructureMetricStrategy
The name of a class implementing *IStructureMetricStrategy*. This is also used by the

TVectorizationStage stage to determine the distance between behaviour models, but based on their structure.

Value: *org.iobserve.analysis.clustering.filter.similaritymatching.GeneralStructureMetric*

sm.IModelGenerationStrategy

The name of a class implementing *IModelGenerationStrategy*. This is used by the *TModelGenerationStage* stage to generate representative behaviour models for groups of behaviour models.

Value: *org.iobserve.analysis.clustering.filter.similaritymatching.UnionModelGenerationStrategy* **or** *org.iobserve.analysis.clustering.filter.similaritymatching.IntersectionModelGenerationStrategy*

sm.parameters.radius

A positive, unitless floating point value. Used by the **TGroupingStage** stage to determine how close two behavior model's vectors have to be to belong to the same group with respect to the parameter distance function.

Value: *At least 0*

sm.structure.radius

A positive, unitless floating point value. Used by the *TGroupingStage* stage to determine how close two behavior model's vectors have to be to belong to the same group with respect to the structural distance function.

Value: *At least 0*

triggerInterval

A whole, positive number interpreted in milliseconds. Used by the *TimeTriggerFilter* stage to determine the interval in between triggering the processing of all behaviour models.

Value: *At least 0*

sink.baseUrl

A String used by *BehaviorModelSink* to determine the output path for models. For windows-style paths, the backslashes should be doubled.

Value: *Any valid filepath*

Chapter 4

Evaluation

4.1 Evaluation Design

We evaluated our results in two stages. First, we drew upon previous work by [Dornieden 2017] and used their example data to run our analysis and compare our results to theirs. Then, we originally planned to use a different set of data to see how our method worked with more complicated inputs. Due to time constraints, we only completed the first evaluation.

4.1.1 Simple User Types

User Behavior Data

For evaluating their user behaviour classification approach, [Dornieden 2017] created 7 user types, characterized by the actions they perform in the JPetStore application. These were designed so that some of the types had overlapping behaviour, while others were entirely unique. They created a workload where they simulated the behaviour of each user type 20 times and captured each execution as a single session. The following are the types that they used, and that we will use for comparison:

Fish Lover
: This user goes to the category *FISH* and adds 8 items of type *FI-SW-01* to the cart. Then, they sign in, go back to the same category and add one more item of the same type. After that, they go to checkout, create a new order and open the order view.

Cat Lover
: This type is functionally identical to *Fish Lover*, but they view the category *CATS* and purchase an item of type *FL-DSH-01*.

Single Fish Buyer
: This user goes to the *FISH* category, views the product *FI-SW-01*, and adds the first listed item to the cart. Then, they log in and order the item from their cart. They use the same transitions as the previous two types, but without multiple edge counts.

Single Reptile Buyer
This user is functionally identical to *Single Fish Buyer*, but they view the category *REPTILES* and purchase an item of type *RP-SN-01*.

Browsing User
This user does not purchase any items, but merely views the store inventory. They navigate to the *REPTILES* category and view the product *RP-SN-01*. Then, they continue by browsing different products and items in the *BIRDS* category. They search for fishes after visiting the category four times, find one fish product and item, and end the session.

New Customer
This user registers as a new customer. Then, they log in with their new account and buy an item from the product *RP-SN-01*, which belongs to the *REPTILES* category.

Account Manager
This user logs in and changes their contact information. Then, they inspect one of their previous orders.

Comparison Data

The approach of [Dornieden 2017] used the X-means clustering algorithm [Pelleg, Moore, et al. 2000]. They ran it multiple times with varying parameters and then picked their best fit based on the least sum of squared errors for the resulting cluster centers. Their best fit generated 6 models, one less than the amount of user behaviours used as input.

Design for simple user type evaluation

To compare our approach to [Dornieden 2017], we set two goals: First, to generate our own best match for the input data, and then to compare it to their best match. Available to us were a tool within the iObserve analysis project [*iObserve Analysis Repository*] to compare behaviour models and list the differences in nodes and edges. We modified this tool to work with our new behaviour model implementation.

Goal 1
Generating user behaviour models that fit the input user types as closely as possible by trying different parameters for our approach and selecting the best result. The parameters we vary are the similarity radii and the model generation strategy.

Question 1.1
For which parameters do our generated models fit most closely with the input data?

Metric 1.1.1: Similarity
By counting the individual differences between models such as different nodes, edges

and edge numbers and adding them, we get a measure of which nodes and edges are missing, and which are additional.

Goal 2
Comparing our best match with the best match of [Dornieden 2017] and deciding which one fits the input data better.

Question 2.1
Which method's results fit more closely to the input data both in terms of number of models and features between those models?

Metric 2.1.1: Similarity Ratio
By counting the individual differences between models such as different nodes, edges and edge numbers and adding them, we get a measure of which nodes and edges are missing, and which are additional.

4.1.2 Fuzzy User Types

User Behaviour Data

The user types in Section 4.1.1 show no variation in their behaviour individually. For a more rigorous test of our method, we planned to run it with input data that is based on the 7 user types used by [Dornieden 2017], but has slight differences on each execution. This means that we have 7 sets of behaviour that are each considered similar within the set but distinct to each other. Unfortunately, due to time constraints, we could not perform an evaluation with this data.

4.2 Realization

In order to find the best fit with the baseline data, we had to first find a good range for the input parameters to run the evaluation with. Since the ideal outcome would be seven generated models, a good indicator was whether a certain parameter range generated models close to that amount. A single run of the analysis indicated that the maximum distance for our test data for the *parameter distance function* was 16.8. To find a range where the amount of models approached seven for the parameter radius, we set the structure radius to 2, which is above its maximum possible value. That way, only the parameter radius would influence which group a model belonged to. We then varied the parameter radius between 5 and 16.8.

As shown in Figure 4.1, the amount of generated models decreases as the parameter radius increases, which is expected, since this means that models are likelier to be grouped together. For the value of 16.8, the minimum amount of multiple generated models is 25. We were unable to find a parameter radius above 16.8 for which less than 25, but more than 1 models are generated. For this, we added increasingly small powers of $\frac{1}{10}$ to the

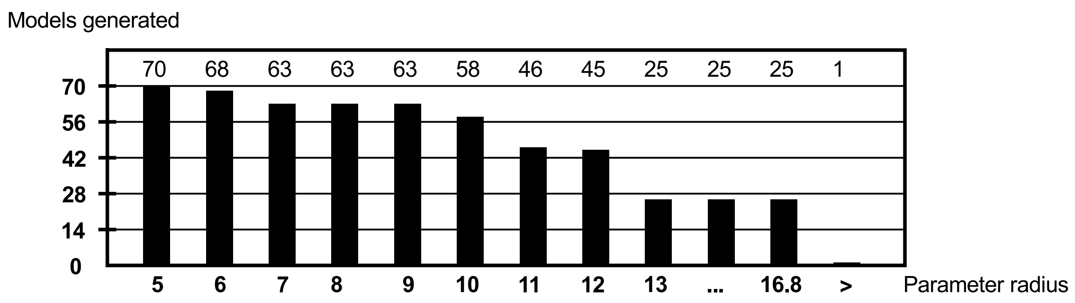

Figure 4.1. Amount of models generated by varying parameter radius between 5 and 16.8 while keeping structure radius maximal

radius. From parameter radius $16.8 + \frac{1}{10}^{14}$ to $16.8 + \frac{1}{10}^{15}$, the amount of models generated changes from 25 to 1.

Due to the inability of our method to generate an appropriate amount of models close to 7 for this data, we decided to evaluate the models generated by both generation methods for the parameters with the minimum amount of models generated, i.e. 25.

Upon further examination, a large part of the generated models were identical. The expectation so far had been that the resultant models must be dissimilar, because similar models should have merged in the group process. This is why we did not think to include a stage to merge resultant models. After manual comparison, the models generated by both strategies each merged into 3 different models.

Two of these models, one from each generation strategy, were identical in structure and call information to the *Browsing User* type. Since this happened for both model generation strategies with the same group, we can conclude that the group this model was generated for consisted only of the *Browsing User* behaviour models, as any structurally different models in that group would have led to the strategies generating a different representative model. Another two models, one of each generation strategy and both of the same group, were highly similar to the *Account Manager* type. Both differed in that they had one missing edge, but two additional edges. Of the remaining models, one model from the union strategy is fairly similar to the * *Lover* types with no missing nodes and one missing edge, but 5 additional nodes as well as 13 additional edges. It it also the most similar to the *Single * Buyer* types, but with 19 missing edges, and 37 additional edges each. The last model, from the intersection strategy, is most similar to the *Account Manager* type with 4 missing nodes and 6 missing edges, but not particularly similar to any other type like the other models are.

These results are worse than the results of [Dornieden 2017], whose best fit generated 6 models. Out of these models, most were highly similar or identical to the base types, with the exception of one model which merged the *New Customer* and *Single Reptile Buyer* types. Our only identical fit, the models that match the *Browsing User* type, are found just as well by their approach. All of our other models are less similar to the baseline models.

Chapter 5

Related Work

[Vögele et al. 2016] introduces an approach they call WESSBAS, which stands for *Workload Extraction and Specification for Session-Based Application Systems*. Their method aims to automate the extraction and specification of workloads to be used for performance testing, reasoning that manual workload specification is both time-consuming and not as accurate as workloads generated by using real monitoring data of user behaviour. They criticize the lack of approaches combining the extraction and specification, claiming that most approaches separate these steps, which results in additional effort. WESSBAS used a domain-specific language to model workload specification in a system-agnostic manner. The algorithm they use to cluster user behaviour is X-Means [Pelleg, Moore, et al. 2000].

In their master's thesis, [Peter 2016] implements their approach for creating models of workload characteristics in iObserve based on monitored user sessions. They use the X-Means algorithm [Pelleg, Moore, et al. 2000] for their clustering and generate usage models of the Palladio Component Model [Peter 2016].

[Dornieden 2017] extended the user behaviour clustering in iObserve by expanding the monitored user sessions by call information. This made a finer distinction between user behaviour possible. Users could now be characterized not only by the actions they take, but also by specific parameters tied to those actions, e.g. what item they purchased. Their approach also used the X-Means algorithm [Pelleg, Moore, et al. 2000] to cluster user behaviour.

Building on this, [Jung et al. 2017] used the Expectation-Maximization algorithm as well as X-Means to cluster user behaviour in iObserve. The EM-algorithm performed worse than X-Means. They mention the problem that certain clustering algorithms have with high dimensionality, and suggest trying different methods in the future.

Chapter 6

Conclusions and Future Work

Our method failed to recognize most user behaviour during our evaluation, performing worse than the approach of [Dornieden 2017]. Due to the flexibility of Similarity Matching, requiring two distance functions, a model generation strategy as well as the radii for the respective distance functions, it is not entirely possible to say whether the unsatisfactory performance can be attributed to a failure of the idea itself - comparing models based on distance functions - or a specific part of the implementation. The fault may lie with the distance functions used, especially the *parameter distance function*, or could even be due to a bug. In order to evaluate Similarity Matching further, more work needs to be done in trying different distance functions, model generation strategies and possibly extending the algorithm to merge similar models after generation.

6.1 Technical Contribution

We implemented Similarity Matching in iObserve [Hasselbring et al. 2013] so that it is easily configurable to use different distance functions and model generation strategies. By creating a class implementing the corresponding interface and including it in the build, changing these elements is as simple as changing the configuration file of the analysis.

The *TSimilarityMatching* stage that contains the actual Similarity Matching is also interchangeable through configuration. It implements the *IClassificationStage* interface, which provides an input port for *UserSession* objects and an output port for an array of *BehaviorModel* objects. Future methods intended for user behaviour aggregation or similar tasks in iObserve can make use of the stage configuration we have provided.

6.2 Future Work

Further testing with different distance functions may show more promising results in the future. The evaluation especially depended on the *parameter distance function* for JPetStore, for which different ideas could be tested. The result that a large part of the generated models were duplicates may be due to a bug, but it could also be an inherent flaw of Similarity Matching. Adding another stage, which merges models that are equal or similar, could be a solution to this problem.

The model generation strategies proposed in this thesis are both on the end of two extremes - combining all of the features, and only taking those features that are shared by every model in its group. A hybrid approach to model generation with its own parameters could be implemented in the future, somewhere between the union and intersection method, where features present in a certain percentage of models are included in the representative model.

Bibliography

[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 1–4, 6, 11, 15, 21–25, 27)

[*HashMap Javadocs*]. *Hashmap javadocs*. URL: https://docs.oracle.com/javase/8/docs/api/java/util/HashMap.html. (Cited on page 10)

[*HashSet Javadocs*]. *Hashset javadocs*. URL: https://docs.oracle.com/javase/8/docs/api/java/util/HashSet.html. (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*. Forschungsbericht. Kiel, Germany: Christian-Albrechts-Universität Kiel, Oktober 2013. URL: http://eprints.uni-kiel.de/22077/. (Cited on pages 1, 5, 6, 27)

[*iObserve Analysis Repository*]. *Iobserve analysis repository*. URL: https://github.com/research-iobserve/iobserve-analysis. (Cited on pages 10, 14, 22)

[*iObserve Webpage*]. *Iobserve webpage*. URL: https://www.iobserve-devops.net/about/. (Cited on page 4)

[*JPetStore GitHub Webpage*]. *Jpetstore github webpage*. URL: https://github.com/mybatis/jpetstore-6. (Cited on pages 2, 6, 7, 11)

[Jung et al. 2017] R. Jung, M. Adolf, and C. Dornieden. Towards extracting realistic user behavior models. *Softwaretechnik-Trends* (Dezember 2017). URL: http://eprints.uni-kiel.de/40365/. (Cited on page 25)

[Kephart and Chess 2003] J. O. Kephart and D. M. Chess. The vision of autonomic computing. *Computer* 36.1 (Jan. 2003), pages 41–50. URL: http://dx.doi.org/10.1109/MC.2003.1160055. (Cited on page 5)

[Moon 1996] T. K. Moon. The expectation-maximization algorithm. *IEEE Signal processing magazine* 13.6 (1996), pages 47–60. (Cited on page 1)

[Pelleg, Moore, et al. 2000] D. Pelleg, A. W. Moore, et al. X-means: extending k-means with efficient estimation of the number of clusters. In: *ICML*. Volume 1. 2000, pages 727–734. (Cited on pages 1, 22, 25)

[Peter 2016] D. Peter. Observing and modeling workload characteristics of dynamic cloud applications. Masterarbeit. Department of Informatics Institute for Program Structures and Data Organization (IPD), 2016. (Cited on page 25)

[Steinbach et al. 2004] M. Steinbach, L. Ertöz, and V. Kumar. "The challenges of clustering high dimensional data". In: *New directions in statistical physics*. Springer, 2004, pages 273–309. (Cited on page 1)

[*TeeTime Webpage*]. *Teetime webpage*. URL: http://teetime-framework.github.io/. (Cited on pages 2, 4)

[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. URL: http://eprints.uni-kiel.de/14418/. (Cited on pages 4–6)

[Vögele et al. 2016] 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* (Oktober 2016). URL: http://eprints.uni-kiel.de/34386/. (Cited on page 25)

[Wulf et al. 2014] C. Wulf, N. C. Ehmke, and W. Hasselbring. Toward a generic and concurrency-aware pipes & filters framework (2014). (Cited on page 4)