Knowledge-Driven User Behavior Model Extraction for iObserve

Master's Thesis

Christoph Dornieden

June 14, 2017

Kiel University
Department of Computer Science
Software Engineering Group

Advised by: Prof. Dr. Wilhelm Hasselbring
Dr. 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, 14. Juni 2017

Abstract

Modern cloud-based software systems are exposed to constant alterations due to changing requirements. These changes can be based on various reasons. Ever-changing usage patterns of the systems user-base can be one reason for changes. A shift in the user-behavior can result in increasing load on single services or have strong economic effects for ecommerce platforms. If the usage patterns are known, the software system can be analyzed and adapted.

In this thesis we provide an approach to extract user-behavior from monitored operation-calls of a software system. Since user-behavior is not only dependent on the navigational pattern of the user in the system, but also on the specific information processed by the call, our focus lies on adding this information to the monitoring records. We then propose an identification of user-behavior models and clustering process for similar user-behavior patterns. These user-behavior models are using the enriched call information to improve accuracy.

We implemented our approach as a pipe-and-filter based service, which is integrated into the iObserve framework. The framework provides access to the monitoring records of the system and transforms them into user-sessions. To get the call-information from the system, we extend the monitoring records to hold this data. In our service, we prepare the sessions containing user-behavior and call-information for the clustering. Then the sessions are aggregated to behavior models by a clustering algorithm.

Contents

Chapter 1

Introduction

A modern cloud-based software system is exposed to constant changes. Continuously services are added, removed, or modified due to changing requirements. A change of a requirement can originate in various reasons. One of this reasons is the change of usage patterns. Over the time the user-base can change and utilize the system differently than it was originally intended. Knowing the user-behavior of a software system allows us to predict future behavior and measure changes, which supports the adaptation and evolution of the software system. For the underlying system this information can be used to predict the number and type of accesses. Adapting the system based on that knowledge can help to decrease the response times and thereby increase the performance for the user. Additionally, knowing the user types of the system can help simulating user access on the system. Load tests become more realistic since users simulation can be based on behavior of real users.

User-behavior can be represented in a directed graph, where all nodes are operations the user called and all edges denote the order in which these transitions are called. In Figure 1.1 an example user-behavior in a systems service is depicted. A user starts using the service by calling operation 1 and follows the path. He can decide how often he takes the loop $(2, 3, 4)$ and which path he takes to reach operation 7. The behavior of a specific user of the service therefore is a sub-graph of the graph in Figure 1.1.

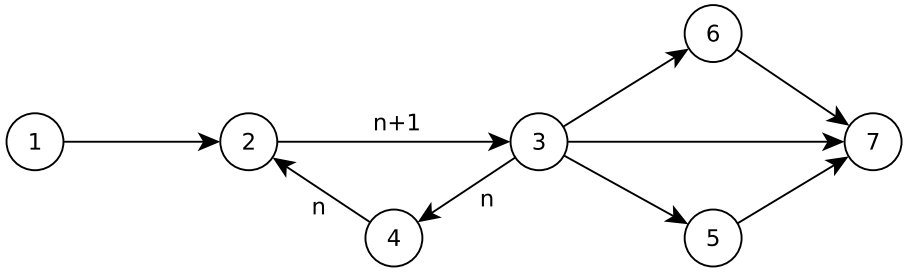


Figure 1.1. Graph a) general graphs showing all possible paths

Figure 1.3, Figure 1.2, and Figure 1.4 each show a specific user of the service. Each user is taking different path from $Node3$ to $Node7$ or a different number of iteration in the loop $(Node2, Node3, Node4)$. These behavior-graphs can be aggregated into one user behavior

model representing both graphs. Abstract behavior models are more manageable than separate behavior graphs, i.e. analyses on small set of behavior-models are faster than on a huge set of all possible behavior-graphs. We aggregate user behavior graphs by comparing all nodes and transitions, followed by automatic grouping of the graphs with the highest similarity.

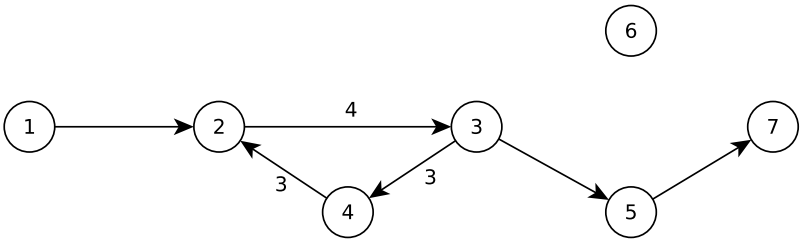


Figure 1.2. Graph c)

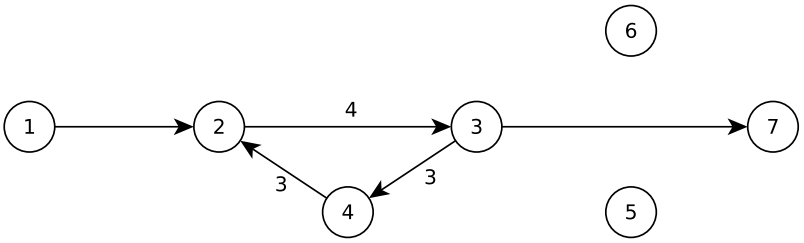


Figure 1.3. Graph b)

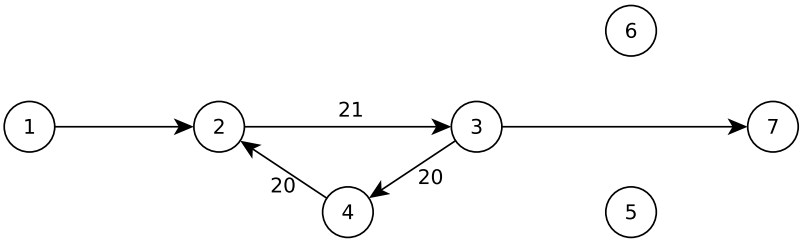


Figure 1.4. Graph d)

If we look at the behavior from Figure 1.3, Figure 1.2, and Figure 1.4, it is not trivial to decide which graphs we should merge into one behavior model. Figure 1.2 and Figure 1.3 have the same number of iterations in the loop but a different transition from *Node*3 to *Node*7, whereas Figure 1.3 and Figure 1.4 have the same transition structure, but a large difference in the number of iterations of the loop. If we compare Figure 1.2 and Figure 1.4, we only find the partial structural similarity, that all graphs of the service share. Thereby, we can not merge all three graphs in one user group. It is hard to decide how we would merge these user-behavior graphs solely based on the graph itself. A solution for this problem is to add user specific data to the nodes, i. e. values of input forms, to describe the user behavior and support the merging. This thesis provides an approach to use this user specific data to improve merging results, integrated in the *iObserve* framework [1] [Hasselbring et al. 2013].

The iObserve project aims to facilitate adaption and evolution of distributed software systems [Hasselbring et al. 2013]. The framework is able to collect user-behavior data, but is missing a component to process this data based on the user-behavior and additional user-specific data. The user-behavior of iObserve is stored as a list of operation calls for each user-session. The contribution of this thesis is to provide a solution to enrich this user-behavior with additional call-information and aggregate the behavior to behavior-models representing user groups.

1.1 Goals

In the following sections we define our two goals for this thesis. The first goal in Section 1.2 is to extract user-behavior models from user sessions enriched with call-information. The second goal is to evaluate our approach, which is defined in Section 1.3

1.2 G1: Extract User-Behavior Models from User-Session

Our first goal is to develop a service which aggregates given user-sessions extended with user specific data to a user behavior model. This service has to be integrated as a *TeeTime* [*TeeTime Framework Webpage*] Filter into the *iObserve* [Hasselbring et al. 2013] framework. It is divided into the following subgoals.

1.2.1 G1.1: Provide a Concept to Enrich User-Sessions with User-Specific Data

The user-sessions of the iObserve framework are a list of entry-calls, which represent a users operation call within the system. These entry-calls are not ale able to store additional data. Since we want to aggregate user behavior based on user-behavior and user-specific data, we have to provide a concept to enrich the entry-calls with additional call-information.

[1] https://www.iobserve-devops.net/

1.2.2 G1.2: Aggregate Extended User-Sessions to Behavior-Models

To extract user groups from user-sessions, we need to aggregate the nested behavior from the sessions. We will solve this task by providing a service to cluster user-sessions and is integrated into the iObserve framework. The service has to fulfill three task. The first task is to preprocess the user-session to be appropriate for the clustering. The second task is to aggregate the user sessions by clustering and the third is transform the clustering results into sufficient behavior models.

1.3 G2: Evaluation of the Approach

Our second goal is to evaluate our approach. Therefore, we will firstly verify in Section 1.4 that our approach is able to cluster enriched user-sessions by finding predesigned user-types in generated workload. Then we will compare our approach with the user-behavior aggregation approach of David Peter [Peter 2016] in Section 1.5 to show that we can match his clustering implementation.

1.4 G2.1: Find Predesigned Use-Types by Clustering Extended User-Sessions

In our first evaluation goal, we want to proof that our approach is able to cluster extended user behavior. Thus, we will create predesigned user types, whereas some are only distinguishable by their call-information and others by their navigational structure.

1.5 G2.2: Comparative Evaluation of the Approach

The second sub-goal is to compare our approach with the approach of David Peter [Peter 2016], who implemented a clustering service for iObserve, which clusters user behavior without additional information.

1.6 Document Structure

This thesis is structured as follows. In Chapter 2 we provide the foundations for this work. It introduces the concepts, frameworks and technologies we used to develop our approach. In Chapter 3 discuss our approach to reach the first goal of this thesis. Then we depict our implementation for the clustering of the user-behavior in Chapter 4. Our approach and its implementation are evaluated in Chapter 5. Chapter 7 concludes our work by discussing our approach and its evaluation. In Chapter 6 we introduce other approaches that are related to ours. Finally in Section 7.3 we depict potential future work.

Chapter 2

Foundations

In this chapter we introduce the tools and concepts used in this work. We begin in Section 2.1 with the pipe-and-filter framework TeeTime, which is used to process our data streams in iObserve. Then we present the Weka data mining tool in Section 2.2.1 and the K-Means algorithm in Section 2.2.2. In Section 2.3 we present an introduction of the iObserve framework, which is the framework our approach is integrated in. It is followed by WESSBAS in Section 2.5, an approach our work will build on. Finally we provide some foundation for our evaluation in Section 2.6.

2.1 The Pipe-and-Filter Framework TeeTime

The *pipe-and-filter* architectural style is used to divide complex tasks into separate executed operations [Buschmann 1998]. Each operation is performed by a segment called *filter*. Filter are connected via *pipes*. A pipe forwards data from one filter to another without processing it. Pipes and filter can be connected to complex structures, since a filter can have more than one incoming and outgoing pipe. That way structures like loops and branches can be created. A pipe-and-filter architecture can therefore easily handle many data of the same kind or so called data streams and process it one date after the other. An example for a simple pipe-and-filter concept in Java would be Java Streams, which are introduced in Java 1.8 [Urma 2014].

The iObserve analyses processes streams of monitoring records and can therefore be appropriately supported by a pipe-and-filter framework. iObserve utilizes the TeeTime framework [Wulf et al. 2014] [*TeeTime Framework Webpage*] to fulfill this task. TeeTime is written in Java and able to realize complex pipe-and-filter structures. It provides developers with broad predefined interfaces and implementations of pipes and filters. The filters of TeeTime are called *stages*, while the pipes keep their name.

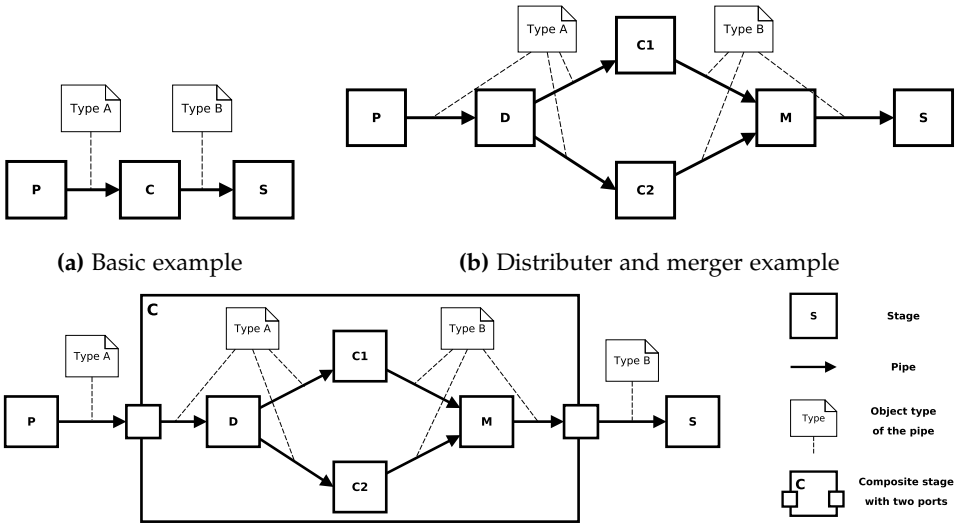
Each stage can have multiple in- and output ports. A port is the interface of a stage. An input port receives data objects and the stage collects it one by one. Processed objects are send to the output port by the stage. Ports are typed and therefor, accept only objects of a certain type. We categorize stages into *producer*, *consumer* and *sink* stages. If a stage has only output ports, it is called producer. On runtime the producer is generating elements and sends them to its output port. A consumer has at least one input port and can have one or more output ports. If an object is located at the input port of the consumer, its execution is

triggered. The object is processed and can be send to one or some of the output ports. A consumer stage with no output port is called sink.

The output ports and input ports can be connected via pipes, whereby each port can only be connected to one pipe at the time. Stages and pipes are easily extensible for various purposes. To create a pipe-and-filter structure with TeeTime, a *configuration* is needed. In a configuration the used stages and the connections between them are defined. The pipes between the stages are then created on runtime. Depending on the configuration, pipes are constructed differently, meaning capacities and transmission type can vary.

Figure 2.1 depicts different TeeTime configuration examples. Figure 2.1d depicts the notation for the configurations in Euclidean Distance, Manhattan distance, and Composite stage example. Stages are plotted as boxes labeled with their names. Pipes and ports are merged to an arrow indicating the direction of the object flow. The type of the in- and output ports is connected to the pipes via a dotted line. The composite stages are boxes containing the inner stages. To illustrate their encapsulation, the ports of the composite stages are smaller boxes on the side of the composite boxes. Ports on the left are always input ports and ports on the right are always output ports.

Figure 2.2a shows a basic configuration. A producer stage `P` produces elements of type `A`, the stage `C` processes them to elements of type `B` and sends them to a sink. In Figure 2.2b the consumer stage `C` is separated into two distinct stages `C1` and `C2`. Both stages transform type `A` elements to `B` elements. A distributer stage `D` is deciding which element is send



(a) Basic example

(b) Distributer and merger example

(c) Composite stage example

(d) Notation legend

Figure 2.1. Examples for TeeTime Configurations

to which consumer stage. The elements of `C1` and `C2` are then merged by merger `M` and forwarded to the sink. Figure 2.2b demonstrates the characteristic of TeeTime, that it is not possible to trigger an execution of consumer stage by two input ports. Each stage can only have one active input port. If an object lies on a passive port, the execution will not be triggered. Therefor, we need a merger to merge the input objects to one stage.

If we want to split a data stream, we either need a distributor or another output port. The same rule applies if we want to obtain elements from two input pipes. In Figure 2.1c we see a composite stage `C` containing a distributer, two stages, and a merger. In TeeTime we can structure stages into groups for a better oversight by wrapping parts of the configuration in container stages.

We will use the visualization notation from Figure 2.2a in the following thesis with a small exception. For the reasons of readability, we will not show distributors and mergers. Every time a stage has more than one input pipe, the deployed architecture contains a merger stage for merging the inputs. The same can be applied for outputs and distributer stages.

2.2 Data Mining Tools and Algorithms

For the aggregation of the user-behavior graphs we will use data mining techniques such as clustering. Therefor, we present in the following section the data mining tools and algorithms relevant for this work.

2.2.1 Data Mining with Weka

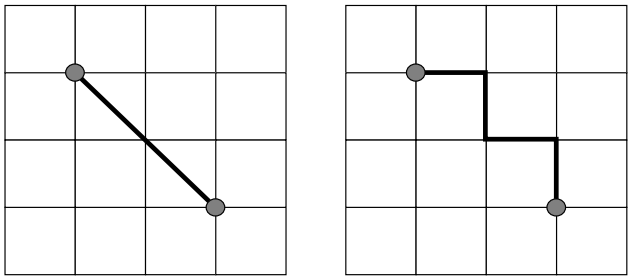
Weka [Hall et al. 2009] is a collection of data mining algorithms provided by the University of Waikato New Zealand. It comes with a GUI tool to process datasets with different data mining techniques like clustering, classification or regression. Also, Weka provides its algorithms as a Java library. To use a Weka clustering algorithm, the input data has to be converted into a special format called *Attribute-Relation File Format*(ARFF). An ARFF file contains a set of vectors and a list of attribute names. The list maps each attribute name to a position in the vectors. The vectors contain the values of the attributes. Each vector is representing another measurement.

Example: If we want to cluster people by weight and height, the attribute list of the ARRF file would contain the two attribute names *weight* and *height*. Each person in our clustering would be represented by a two dimensional vector. The first attribute in the vector would be the weight of the person and the second its height.

The Java implementation of the ARFF is called `Instances`. An `Instances` contains the list of attribute, the vectors of attribute values are stored in `Instances` objects.

2.2.2 The Clustering Algorithms K-Means and X-Means

For our approach we will use the X-Means [Pelleg and Moore 2000] clustering algorithm of Weka. It is a special variant of the K-Means algorithm [DU 2010, section 4.3]. The K-Means receives as input a set of vectors $v_1 \dots v_n$ and a number k. The number k represents the number of desired clusters. The K-Means chooses k random vectors as cluster centers and adds the other vectors to the clusters based on a distance metric. A distance metric defines the distance between two vectors, since there is more than one way to calculate the distance between two points. Popular examples are the *Manhattan* or the *Euclidean* distance [DU 2010, section 4.2.3] which are displayed in Figure 2.2. The euclidean distance is the direct distance between too data points, while the Manhattan distance is adding distances in each dimension. Depending on how important the comparison of inner-dimensional values in comparison to inter-dimensional values are, we can choose one over the other.



(a) Euclidean Distance **(b)** Manhattan distance

Figure 2.2. The Manhattan and the euclidean distance

The K-Means clustering is performed like followed. First we either pick k vectors from the vector set or create k vectors randomly and call these vectors centroids. At the end of the clustering each centroids will be the mean vector of a cluster. Then we assign each vector v_i to the nearest centroid c_j based on a given distance metric. After the assignment of all vectors to a centroid, we set the mean vector of all vectors in the group as new centroid. The next step is, we iterate over the procedure by starting with assigning all vectors to their nearest centroid until the mean vector of the vector group is almost identical to the centroid. Then every group is a cluster and every mean vector is its centroid.

One quality measurement for the quality of a cluster is the Sum of Squared Errors (SSE). It sums up the distance from all points to its cluster centers. It describes the density of the clusters. The smaller the SSE is, the better is the clustering. Figure 2.3 shows the formula of the SSE, where k is the number of clusters, C_i are the clusters with its centroid m_i, and x is a vector in this cluster. The function *dist* computes the distance of two vectors based on an arbitrary distance metric.

$$SSE = \sum_{i=1}^{k} \sum_{x \in C_i} dist(m_i, x)$$

Figure 2.3. Sum of Squared Errors (SSE)

K-Means is designed to find a specific number of clusters, but sometimes it is hard to guess how many clusters can be found in a vector set. X-Means therefore executes K-Means for a range x and returns the result with the most fitting clusters. The input for the X-Means are a minimum, and a maximum number of clusters (k_{min} and k_{max}) and the input vectors. The algorithm is divided into three steps.

1. **Improve-Parameters**
2. **Improve-Structure**
3. If $k > k_{max}$ stop and return best solution found. Else, Goto 1.

Improve-Parameters In the first iteration of X-Means, the K-Means clustering algorithm is executed on the vectors for k_{min}. In every other iteration the K-Means is executed on current state with the current k. The result of this execution is then saved for further use.

Improve-Structure In the second step, it is examined if one or more of the current cluster centroids can be split into two separate centroids. Therefor, each cluster is considered as an own clustering instance with a single centroid. The centroid of each instance is split into two child centroids. The new centroids have the same distance d to the position of the parent centroid and the distance $2d$ to each other. We execute the K-Means algorithm next with $k = 2$ on each instance. Then the cluster of the children is compared to the cluster of the parent. If the child centroids perform better in the evaluation than the parents, we keep them by removing the parent and adding the children to the global instance. Thereby the k is increased. If the children do not perform better than the parent, the cluster stays the same.

When $k = k_{max}$ the maximum number of clusters is reached. The K-Means is executed a final time for the k_{max} instance and then the found clustering results for all k are compared. Finally the best result is returned.

2.3 The iObserve Approach

The iObserve approach (Integrated Observation and Modeling to Support Adaptation and Evolution of Software Systems) [Hasselbring et al. 2013] [Heinrich et al. 2015] assumes that a modern software system is exposed to constant deployment changes due to varying usage and requirements. The iObserve approach is designed in order to support evolution and adaptation and to react to these conditions. Evolution in this case means manual modification of the system made by developers and operators, while adaptation means automated improvements. Figure 2.4 visualizes this concept by showing the iObserve approach. It follows the *MAPE-K* approach [Kephart and Chess 2003]. A *MAPE-K* framework monitores a system or parts of a system. The resulting data is analyzed. If the system changed, a plan is created to optimize the changed system. This plan is then executed on the monitored system.

In iObserve the planning and execution is divided into two parts. The first is called *adaptation*. The adaptation is an automated change of the the system based on a planning algorithm. The second is called *evolution*. The evolution operators evaluate the monitored and analyzed data and realize changes manually.

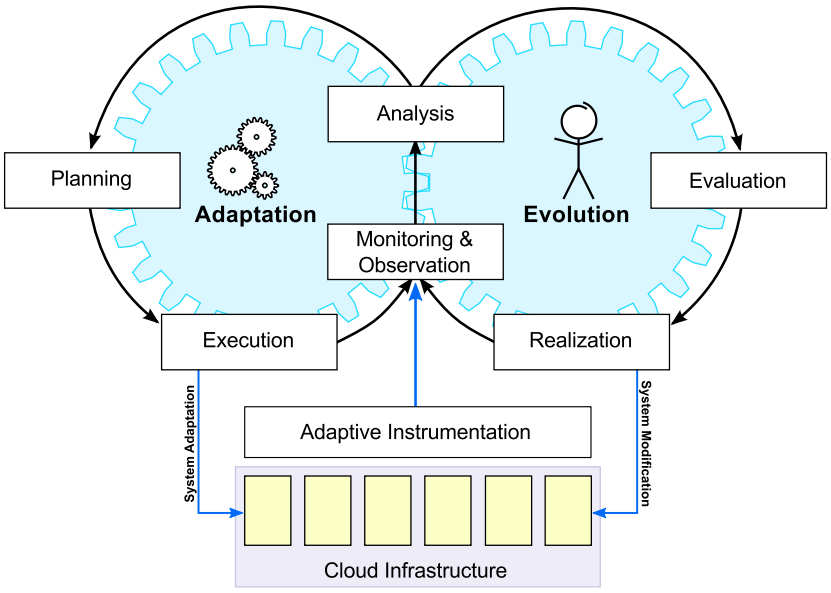


Figure 2.4. iObserve approach

2.3.1 The iObserve Monitoring Component

The iObserve approach uses the Kieker monitoring framework [Van Hoorn et al. 2012] as base for its monitoring. The Kieker framework supports instrumentation technologies and provides monitoring, which is necessary to observe runtime properties, like control-flow tracing and application performance. When a service instrumented with Kieker is accessed, the monitoring creates one or more *Kieker monitoring records*. The records are containers for data gathered from the service. Each property in these records represents a single measurement and can be represented by Java primitives and strings. Monitoring records are collected by the *Monitoring Controller* and stored or sent via a *Monitoring Writer*. In iObserve monitoring data is either sent to a remote analysis server via network stream or written to a log file for a delayed analysis.

For iObserve, the Kieker instrumentation language (IRL) [Jung 2013] is used to extend the existing Kieker monitoring records, as well as, creating new records. In Listing 2.1 we see an example for an extended monitoring record. The defined `EJBDeploymentEvent` extends the `AbstractEvent` of Kieker. It contains three string properties. An `EJBDeploymentEvent` is created by an instrumented service when a servlet is deployed or undeployed.

Listing 2.1. iObserves EJBDeploymentEvent defined in Kieker IRL

```
1  abstract entity EJBDeploymentEvent extends AbstractEvent {
2    string serivce
3    string context
4    string deploymentId
5  }
```

2.3.2 The Palladio Usage Model

In our evaluation we want to compare our behavior models with the current behavior models of the iObserve approach. The current models are stored as usage models of the Palladio Component Model (PCM) [Becker et al. 2007; 2009]. It is a meta-model to describe software systems. A Palladio instance comprises multiple sub-models, each characterizing a view of system. This sub-models are used in the iObserve framework to map the system to a model. For our approach we need to understand the *UsageModel*, which describes the user-behavior of the system and is used as behavior model in the current user-behavior aggregation approach by Peter [Peter 2016].

Figure 2.5 depicts this usage meta-model. The *UsageModel* is the collection of all user groups of the system. It contains one or more *UsageScenarios*, where each scenario represents a group of users from the system. The user-behavior itself is modeled by the *ScenarioBehavior*. The scenario behavior contains a list of *AbstractUserActions*. Each monitored user action is modeled by an *EntryLevelSystemCallEvent* and can be a *Signature* call, a variable usage, or the user role. This action list always begins with a *Start* and always ends with a *Stop* action.

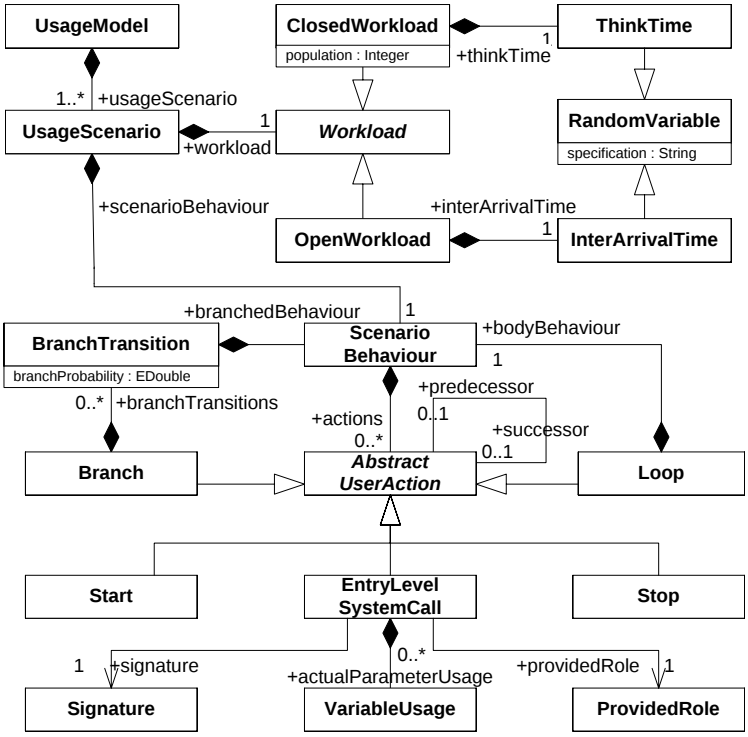


Figure 2.5. Palladio usage model from [Becker et al. 2009]

All actions between start and stop are interconnected, that way all actions can be traversed. This path of user actions is the monitored path of the users through the system. The PCM only allows to concatenate user actions, to make a loop or a branch. The *Loop* and the *Branch* action are necessary to model it. Therefore, a loop contains another behavior scenario, which models the loop body. When traversing down the inner scenario recursively, is the first entry call that is found thereby the start and the end of the loop. Branches contain a set of branch transitions, each holding a probability and an inner behavior scenario. Due to the inner scenarios of loops and branches, the behavior scenario is a recursive structure and measured by the effort of coding.

2.3.3 The iObserve Analysis

iObserve uses models of the monitored system to analyze system properties. The models are specified using the Palladio Component Model [Becker et al. 2009] of the monitored system to analyze the system. Initially a model of the system, including architecture, deployment, and user-behavior is created by a human developer. These models are then updated at runtime based on the analysis of the monitoring data. Based on the model and

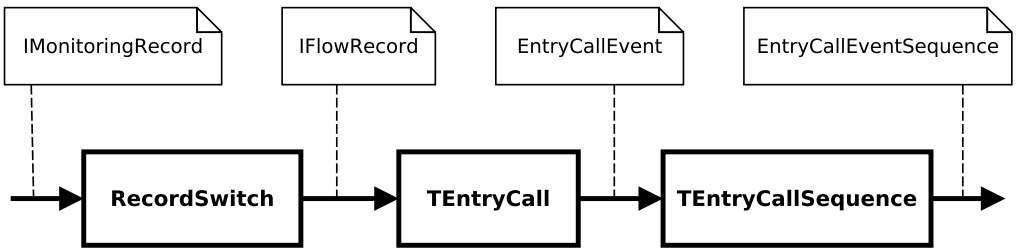


Figure 2.6. Processing of entry calls

the result of the analysis, the model can be modified. If the analysis proposes an adaptations based on the modified model, this adaptation is evaluated and could be realized in the monitored system.

Besides deployment, iObserve also analyses user-behavior models. These models are based on entry call events, which can be synthesized based on call traces. In the monitored system two Kieker records are created and sent to the monitoring writer for each operation call. The `BeforeOperationEvent` is created when the operation is called and the `AfterOperationEvent` when the operation returns. To process these records, the iObserve analysis uses a TeeTime configuration. A reader stage reads incoming records and transforms them into Java objects. These objects implement the `IMonitoringRecord` interface. They are sent to the `RecordSwitch` stage, which switches the records by type and sends them to processing stages for the respective records types. The `BeforeOperationEvent` and `AfterOperationEvent` implement the `IFlowRecord` interface and are, therefore, forwarded to the `TEntryCallEvent` stage. This can be seen in Figure 2.6, which shows a partial configuration of the analysis containing only stages necessary for tracing user-behavior. The stage `TEntryCallEvent` merges `BeforeOperationEvent` and `AfterOperationEvent` of an operation to one record called `EntryCallEvent`. These `EntryCallEvent` records are then sent to the `TEntryCallSequence` stage. At this point, the `EntryCallEvent` records are sorted by user and session. For each user an `EntryCallSequenceModel` is created containing a list of sessions, whereby each session is containing a list of `EntryCallEvent` records.

2.4 iObserve Behavior Model Visualization

The user-behavior model visualization of iObserve [Banck 2017] is web service to visualize user-behavior graphs. It consists of a backend and a frontend component. The core backend component is a Neo4j[1] graph database storing the behavior graphs. It can be accessed via REST to create, update, or delete graphs. It is connected to frontend via websocket. The

[1] `http://neo4j.com/`

frontend is a React[2] app visualizing the graphs from the backend.

In the internal model, the directed graph of the system is called `Application`. Each application contains multiple nodes called `Page` objects. The properties of a page are all transitions. The page is a part of a map of additional properties, which can be added dynamically to the page. The transitions between those pages are called `visits`. A visit has a source and target page. Furthermore, a visit has a property `calls` for the number of transitions from the source to the target.

If we want to add a graph to the visualization, we send a *POST*-request with a JSON of an `Application` to the backend. In return we get an identifier, which enables us to referentiate the application later. It is impossible to create entities with an own identifier on the backend service. After we added the application, we can add first the nodes and then the edges one by one. When we add a node, we have to save the returned identifier to referentiate the nodes in the edges. Since the REST interface does not support adding of listed elements, we have to create a REST request for every entity sent to the server.

2.5 WESSBAS Approach

Like our approach, WESSBAS uses the modeling and automatic extraction of probabilistic workload specifications from session-based application systems [van Hoorn et al. 2008; 2014; Vögele et al. 2015]. WESSBAS uses it for load testing on these systems. It aims to use real user-behavior to generate scalable workload for testing. The approach predicts that using real user-behavior base for the simulation of load on the system will make load test more realistic and, therefore, more effective.

The architecture of this approach is depicted in Figure 2.7. It shows the four parts of the approach. The first part is the monitoring. In the *Monitoring*, the instrumented system under test (SUT) is used to create session logs. In the *Behavior Model Extractor* these session logs are then transformed to an absolute behavior model for each session. This model is a call graph represented by a $n \times n$ matrix containing the absolute transition frequencies of the sessions' operation calls. The matrices are then transformed to vectors and clustered with Weka. The results of the clustering are behavior models, a behavior mix and the workload intensity. The behavior models are represented by Markov chains, where the Markov states are associated with operation calls. The transitions between the states model the probability of a user following the path of the transition. The behavior mix is storing how much percent of the overall user base a behavior model represents. Finally, the workload intensity is a function defining the arrival rates of new sessions over time. The behavior models and the behavior mix are then transformed into WESBASS-DSL instances in the next component called *WESSBAS-DSL Model Generator*. The WESSBAS-DSL is the core of the WESSBAS approach. It is used to define a workload model for load testing of a system. The transformed behavior models are then sent to *Test Plan Generator*, which

[2] `http://facebook.github.io/react/`

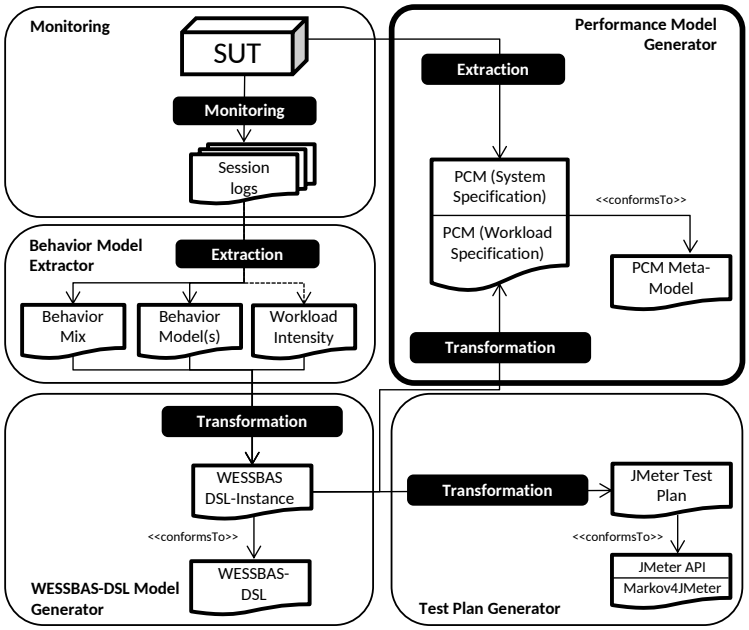


Figure 2.7. WESSBAS approach overview taken from [Vögele et al. 2015]

creates a JMeter test plan to simulate realistic user behavior on the system. In addition to the test plan creation, WESSBAS has a fourth component. The *Performance Model Generator* is using the monitoring data of the SUT and the extracted WESSBAS-DSL instances to create PCM models of the system to store the workload characteristics of the system. Since WESSBAS-DSL can not be mapped directly on any PCM Model, various models are created and missing informations directly extracted from the monitoring.

2.6 Tools Used in the Evaluation

In this section we present a set of tools and projects we use for our evaluation. First, we will describe the JPetStore, a webstore we instrumented to evaluate our behavior model creation, in Section 2.6.1. Then we briefly introduce the Common Component Modeling Example (CoCoME)[Herold et al. 2008] in Section 2.6.2. We will use the application CoCoME in order to compare our approach to the approach of David Peter [Peter 2016]. In Section 2.6.3 we provide an overview of the workload generation for our evaluation. Finally we explain the *Goal-Question-Metric* framework we will use to evaluate our approach.

2.6.1 JPetStore

The JPetStore[3] is a demonstration for a simple web application built with Java. It is built on top of the frameworks Spring[4], MyBatis[5], and Stripes [6]. The JPetStore incarnates a full web shop including account, item, and order management. It is implementing the model-view-controller (MVC) pattern. The controllers of the store are implemented as Stripes `ActionBean`. When we are navigating the site, we are navigating through the different actions. All actions are accessed with the same path.

```
http://localhost:8888/jpetstore/actions/
```

If we want to access the item catalog, we call the `Catalog` action. This is also represented in the URL.

```
{path}/Catalog.action
```

Every sub page containing parts of the catalog, like categories or items, is accessed with the query parameter of the catalog URL. If we want to see the category *FISH* we call the catalog action with the queries `viewCategory=` and `categoryId=FISH`.

```
{path}/Catalog.action?viewCategory=&categoryId=FISH
```

The same concept is applicable for all sites of the store with an underlying model. The only page with no underlying model is the help page.

We use the JPetstore in the version 6.0.2 to evaluate the software we create in our approach. Therefor, we will interpret the the URLs as operation calls.

2.6.2 CoCoME

For the comparison of the two behavior model aggregations, we will use monitored behavior from the Common Component Modeling Example (CoCoME) [Herold et al. 2008]. CoCoME is an enterprise management tool, where there exist different user roles. An enterprise manager, e.g., can create new stores or products for an enterprise. For our evaluation we will use the behavior of a cashier. A cashier uses the cashdesk. He starts a purchase, scans items, and collects the payment of the customers. The actions of the cashier can be mapped directly to the behavior of a customer, e.g. whether a customer buys many items, the cashier scans many items.

[3] `https://github.com/mybatis/jpetstore-6`

[4] `https://projects.spring.io/spring-framework/`

[5] `http://www.mybatis.org/mybatis-3/`

[6] `https://stripesframework.atlassian.net/wiki/display/STRIPES/Home`

2.6.3 Workload creation with Selenium

For our evaluation we have to create workload on the JPetstore and CoCoME simulating multiple different users. [Selenium] is a framework for browser automation. It is primarily designed for testing web applications but is not restricted to it. We can use it to create sequences of web operations representing a certain user-behavior. Therefore, we need to create a script containing a clickstream and run it on a Selenium WebDriver. A WebDriver is an interface for translating commands into operations in a browser. The scripts can be written in different languages like Java, C# or Ruby and run on the respective implementation of Selenium. When a script is executed, opens the driver a browser window and acts as a user.

To create a script we can either write it manually or record our clickstream with the Selenium IDE[7] , a plug-in for the Firefox browser. It can record clicks and save them in html scripts. Scripts can be executed by the IDE to repeat all actions directly or converted to other languages.

To use Selenium with these scripts in our project we need to add the WebDriver Jar to our project. This can be done via Maven (see Listing 2.2).

Listing 2.2. Selenium Web Driver Maven Dependency

```
1  <dependency>
2      <groupId>org.seleniumhq.selenium</groupId>
3      <artifactId>selenium-java</artifactId>
4      <version>3.3.1</version>
5    </dependency>
```

The Selenium community provides drivers for almost all common browsers such as Firefox or Chrome. The browser we will use is the headless browser [PhantomJS]. PhantomJS is a command line browser running headless in the terminal. It is implemented in Javascript using [Node.js]. Selenium offers us WebDrivers for this browser but not the browser itself. Therefore, it has to be installed separately. We use PhantomJS since we do not need a browser window to see how the clickstream is executed. We install Node.js to our system, followed by using the Node Package Manager with the command '`npm install -g phantomjs`'. Now the PhantomJSWebDriver can be instantiated as seen in Listing 2.3.

Listing 2.3. Create a PhantomJs Driver for Selenium

```
1  public PhantomJSDriver createPhantomJSDriver() {
2
3        final DesiredCapabilities capabilities = new DesiredCapabilities();
4        //set path of phantomjs executable
5        capabilities.setCapability(PhantomJSDriverService.
             PHANTOMJS_EXECUTABLE_PATH_PROPERTY,
```

[7] `https://addons.mozilla.org/en-US/firefox/addon/selenium-ide/`, *visited: 20.03.2017*

```
6                 this.PATH_PHANTOMJS);
7         //load javascript on site
8         capabilities.setJavascriptEnabled(true);
9         // able to take screenshots
10        capabilities.setCapability("takesScreenshot", true);
11
12
13        final PhantomJSDriver driver = new PhantomJSDriver(capabilities);
14
15        driver.manage().timeouts().implicitlyWait(30, TimeUnit.SECONDS);
16        driver.manage().window().setSize(new Dimension(800, 600));
17
18        return driver;
19    }
```

Listing 2.3 shows the configuration for the PhantomJSWebDriver. First we provide the WebDriver with the path to our PhantomJS executable. Then we configure our driver to load JavaSript, since almost all modern sites use JavaScript elements. Additionally, the driver is able to take screenshots of the virtual browser window, which will help us debugging.

The PhantomJSWebDriver can be used to browse a site. An example for a script that can be executed is shown in Listing 2.4. It shows how a WebDriver is using a login form to log into a website by navigating the DOM tree, finding elements and calling events on them.

Listing 2.4. Using the PhantomJS WebDriver

```
1   private void browseSiteWithWebDriver(final WebDriver driver) {
2
3         //call the page
4         driver.get("http://localhost:8080");
5
6         //navigate the page
7         driver.findElement(By.linkText("Sign In")).click();
8         driver.findElement(By.name("username")).sendKeys("name");
9         driver.findElement(By.name("password")).sendKeys("password");
10        driver.findElement(By.name("signon")).click();
11
12    }
```

2.6.4 The Goal Question Metric Method

For our evaluation we use the Goal-Question-Metric method (GQM) [Basili and Weiss 1983]. The GQM was originally defined for the NASA Goddard Space Flight Center. It

was created for evaluating flaws of projects and is based on the following assumption. To measure the success of a project, its goal must be specified first with respect to which data and operational methods. The GQM was extended into an application by [Van Solingen et al. 2002]. It results in the specification of a measurement model which defines questions and rules to interpret the measurement data. The model is divided into three levels. These levels are depicted in Figure 2.8.

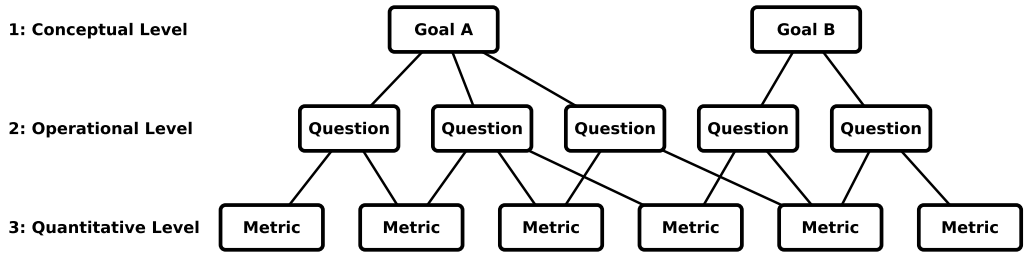


Figure 2.8. Levels of the GQM framework [Basili 1992]

The first level is called conceptual level. It is the goal we want to reach with the GQM method. A goal has a set of properties, which are defined in a template. The goal definition properties are *object of study*, *purpose*, *focus*, *stakeholder*, and *context*. The *object of study* can be either a product, a process, or a resource. The object is analyzed for a *purpose* with a *focus* on a part of this object. The achievement of the goal can depend on different people, called *stakeholder*. In some cases reaching the goal is not only dependent on the *stakeholder*, it is also dependent on the environment the object is located in. This environment has to be defined in the goal as *context*.

The second level is the operational level. It contains a set of questions which are used to describe a quality aspect of the object mentioned before. The questions also imply how the achievement of a goal is going to be conducted. It is preferable that a question can be answered quantitatively.

The third level is the quantitative level. It contains metrics to measure the data provided with the object. With metrics data can be measured in a quantitative way. Metrics are used to answer the questions. Multiple metrics can be used to answer a question and metrics are usable by multiple questions.

A GQM implementation consists broadly of four phases [Southekal 2017], which are shown in Figure 2.9. The first phase is the *planning* phase. In the planning phase, the environment of the object of study is characterized, leading to a specification of all properties of the goal. Then the goal is defined in the *definition* phase. We define questions to specify the goal and metrics to reach our measurement goal. In the *data collection* phase, we collect all data relevant in the environment of the goal. After that we can analyze and interpret it in the *interpretation* phase. We use the measurements done with the metrics to answer the questions. Thereby, we can verify if we accomplished the goal.

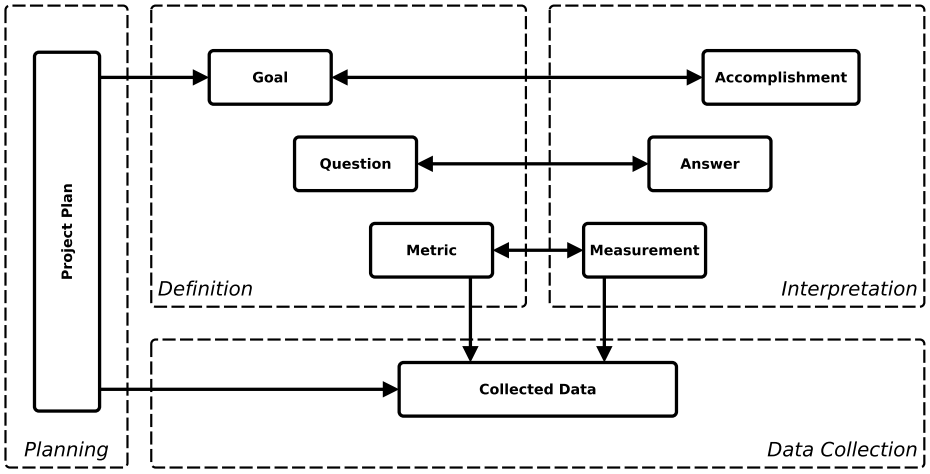


Figure 2.9. Phases of the GQM [Southekal 2017]

Chapter 3

Approach

In this chapter we formally describe our approach for aggregating user-behavior as defined in goal G1 (see Section 1.2). The extraction of user-behavior from session logs is a known problem and different solutions were already proposed [Banerjee and Ghosh 2001; Peter 2016; Dharmarajan and Dorairangaswamy 2016]. One solution is the Model Extractor component of the WESSBASS approach for workload extraction [van Hoorn et al. 2014] as described in Section 2.5. We extend their Behavior Model Extractor approach by adding user specific information. User specific data is defined as all data, that is associated with a user and not represented by the user sessions. It is data such as, input of login forms, parameters of internal operation calls, or general user information.

The current version of the iObserve analysis groups monitored operation calls by session and user. For each distinguishable user, a set of sessions is stored. Each session is a list of operation calls.

In our approach, we take the following steps to extract the user-behavior from monitored user sessions and user specific data. First, we propose our concept for enriching user sessions with additional call information in Section 3.1. Before clustering, we filter all non relevant operation calls to reduce the complexity of the clustered vectors. This is shown in Section 3.2. When we have filtered our sessions we need to prepare them for the clustering. Since we use the Weka library (see Section 2.2.1) for clustering, we have to transform each session with its list of calls to a vector. We explain our concept for this transformation in Section 3.3. After that, we can cluster the vectors, which is explained in Section 3.4. Finally, we provide the definition of our behavior models in Section 3.5.

3.1 Enriching the User Sessions with Call Information

Goal G1.1 in Section 1.1 describes the necessity to merge user-behavior graphs with user specific data. We reach this goal by extending the model for the entry-calls of the current iObserve system. An entry-call models 1an operation call from the monitored system. It is a record type containing different attributes of the call, like the operation signature. We extend this type with the attribute *callInformation, typed as an string* array to hold all passed parameters. The exact implementation is described in Section 4.1.

3.2 Filtering Entry Calls

The set of entry-calls represents the operation calls generated in a system by direct user action. Not all operation calls are relevant to trace the user-behavior. For example, an operation can invoke other operations in the system. If operation a invokes always operation b and c, we are not interested in the call of b and c after we found an operation call for operation a, since we know that b and c will be called. Furthermore, not all entry-calls are relevant either. For example, the navigation of the user is bound to multiple entry-calls. In this case, we can again generally assume that after the call of the first operation, the other operation calls will follow. Operation calls, like in the second example, are not always irrelevant, assuming three operations are normally called after another, if the user uses the system correctly. When the user exits the system after calling the first operation, the second an third operation become relevant. Now they are not part of the call sequence, which they would have been if we declared them as not relevant.

We create a filter function $\delta : E \to \mathbb{B}$, with E being the set of all entry-calls. This function returns *true* if the entry-call is relevant and false otherwise. The relevance of an entry-call has to be decided for each system and, therefore, has to be specified for each system separately by an operator or a developer. Only a developer or operator can decide whether an entry-call is relevant. Such information can also be derived from the design-time model, e.g. the Palladio Component Model.

3.3 Transforming User Sessions to Vectors

Our given system provides us with a set of user sessions $\mathcal{U}$ for each distinguishable user while every session is a list of entry-calls enriched with call-information. For the clustering, we need to transform these list of enriched entry-entry calls into a vector. The vector has to contain each transition of the user session and the call-information of the entry-calls, which are part of the transition. Every vector has to have the same dimensions and has to be equally structured, since the clustering needs the same kind of data at each field of the vector to provide a useful result. Therefore, we divide the creation of the vectors into two steps. First, we create a matrix containing all possible transitions and a function to assign a call-information to each entry call in Section 3.3.1. This matrix is used to create a matrix for the transition of each session. The call-information for each session is collected in Section 3.3.2. In Section 3.3.3 the matrix and the call-information are transformed into a vector for the clustering.

3.3.1 Creating the Transition-Matrix

For the creation of the transition-matrix, we first create a matrix prototype from the entry-call signatures of the user sessions, then we setup an instance of this prototype matrix for each session. Finally, we explain how we connect the call-information to the signatures.

Our given system is providing us with a set of user sessions $\mathcal{U}$ for each distinguishable user while every session is a list of entry-calls. Based on these sessions, we generate a set S containing every operation signature from all existing entry-calls. Then we create a zero $n \times n$-matrix T, with $n = |S|$, for each session, based on all operation signatures. The bidirectional function $o : S \rightarrow \mathbb{N}$ maps every operation call to a row and column in the matrix. The position (i, j) with $i, j <= n$ in the matrix denotes the number of transitions from operation $o^{-1}(i)$ to operation $o^{-1}(j)$ with o^{-1} the inverse function to o. We need the inverse function to be able to lookup the operation signature for an specific index, e.g. such function is useful when we iterate over the matrix.

To add a transition to the matrix and therefore, modifying the transition counts T, we can reference it by first getting the indices for the matrix with the function o by calling $o(s), s \in S$ for the source and the target signature of the transition. The matrix T is built symmetrically, meaning that the index i in a row and in a column always references the same signature. Thus, the $t_{i,i}$ shows transitions from the service $o(i)$ to itself.

To setup the matrix T, we add every transition from the session to the matrix of the session. A transition is a pair of two entry-calls. For each entry-call e_i from the the session u with $0 < i < |u|$, we add the pair (e_{i-1}, e_i) to the transition matrix if the filter function allows each entry-call, i.e., $\delta(e_{i-1}) = true$ and $\delta(e_i) = true$. If one entry-call e_i of a transition is not valid and thereby, $\delta(e_i) = false$, we skip this call and replace it. We replace e_i it with e_{i+j}, where $j \in \mathbb{N}$ $\delta(e_{i+j}) = true$ and for all $0 < k < j$ $\delta(e_{i+k}) = false$.

For the clustering we need our data in the form of vectors. The dimensions of these input vectors should be as small as possible to create the best result in the clustering. Higher dimensions can lead to longer distances between the data vectors. This can be aggravated by the used distance metric. The closer two vectors are, the more likely it is that they are in the same cluster. With higher dimensions the distances become longer and it is harder to find clusters [Steinbach et al. 2004]. Thus, it would be favorable to reduce the dimension of the vector.

Not every transition is possible in a system, thus, there exists a position (i, j) with $i, j \leqslant n$ where $\sum_{S \in \mathcal{S}*} \sum_{s \in S} s_{i,j} = 0$, while $\mathcal{S}^*$ is the set of all sessions. This means there are transition counts that are zero for every session. We call these transitions *empty* transitions. An empty transition increases the dimensions of the vectors unnecessarily. However, it will not affect the result since the value will always be zero, because the distances between vectors are not increased. The computational effort will be increased instead, due to the high number of input values for the distance calculation. We define a function $\lambda : \mathbb{N} \times \mathbb{N} \rightarrow \mathbb{B}$ which returns *true* if the transition is empty and $false$ otherwise.

3.3.2 Managing Call-Information

A call-information is a key-value pair. Multiple call-information can belong to a operation signature, where each call-information key is unique for each signature. Therefore, we create a set $I_{(s,U)}$ for each signature $s \in S$ and user session $U \in \mathcal{U}$. $\mathcal{I}$ is the set of all existing $I_{(s,U)}$. $I_{(s,U)}$ contains a set for every distinguishable call-information key for every entry-call,

that had the operation signature s in the session u. The function $\gamma : S \times \mathcal{U} \rightarrow \mathcal{I}$ maps every signature of a session to its call-information.

3.3.3 Transforming the Transition Matrix into Vectors

We now transform each session matrix to a vector by adding every position (i,j) with $i, j < n$ of the matrix to the vector if $\lambda(i, j) = true$. If the set of all existing distinguished call-information keys is K, we add $|K|$ fields to the vector. This is done because we want to cluster the call-information together with the behavior graphs. We define a function $\delta : K \rightarrow [n \ldots n + |K| - 1]$ which maps every key to a position in the vector. Now we add the value of one call-information c out of every $I \in \gamma(s, u)$, where $s \in S, u \in U$, to the vector at the position $\delta(c.key)$. To find the best c in I we define a function $\beta : \mathcal{I} \rightarrow I$ which returns the best fitting call-information. This function has to be implemented for every system individually, since every system has different call-information. In Section 5.2, we provide an example implementation.

3.4 Clustering of the User Sessions

The created vectors from Section 3.3 are then clustered by a clustering algorithm. In this section we describe which algorithm is used in Section 3.4.1 and how it is applied in Section 3.4.2.

3.4.1 The Clustering Algorithm

For the clustering of the user sessions we use the *X-Means* clustering algorithm [Pelleg and Moore 2000]. As described in Section 2.2.1, X-Means is an extension of the *K-Means* algorithm. Thus, the most characteristics of the K-Means algorithm are also applicable for X-Means. These characteristics are, for example, feasibility and scalability [Suthar and Oza 2015], good results on huge datasets [Verma et al. 2012], and its sensitivity for noise [Abbas 2008]. We use X-Means because it is a clustering algorithm which performs in general with good performance and needs a low configuration effort. Due to those characteristics, we choose it to act as a general purpose solution for a clustering. Since there is no clustering algorithm fitting perfectly for all clustering problems, our approach allows us to exchange X-Means by other clustering algorithms. The implementation of the clustering can be seen in Section 4.3.3.

3.4.2 The Execution of the X-Means Clustering

The higher the range of the X-Means, the higher is the time performance, since the X-Means is executed for every number k in the range $x = [i...j]$, with $i, j \in \mathbb{N}_0$ and $i \leqslant j$. The execution time of X-Means depends on the input range for the cluster sizes. In order to reduce the

time-performance, we have to estimate the cluster range in advance. This can not be done in general, since it always depends on the analyzed system and its user-behavior.

Another input of the X-Means clustering is the. For our clustering we choose the Manhattan distance[DU 2010, section 4.2.3], since we want, for its inner-dimensional distance to have a higher effect on the overall distance than inter-dimensional distances.

We execute the X-Means clustering multiple times and compare the sum of squared errors of each clustering result to each other. Our output is the cluster centroids with the lowest SSE. This is necessary, since the X-Means clustering is very sensitive to the initial centroids [Suthar and Oza 2015]. The initial centroids are chosen randomly, it can happen, for example, that two centroids are placed in one cluster. It is possible that in the end, these centroids define two clusters, splitting the existing one. Vice versa an initial centroid can be placed in the middle of two clusters with no other centroid closer to them. During the clustering, the two clusters will be added to the centroid in the middle of them. The result of the clustering will be one merged cluster.

The result of the clustering will be a set of $k, k \in X$ vectors. Each vector is the centroid of the cluster and each cluster is a group of users. The centroid vectors represent the behavior of all user in the cluster.

3.5 The Behavior Model

The result of the clustering is transformed to our behavior models. A behavior model is a graph $\mathscr{M} = (N, E, \Delta, \Gamma)$. N is the set of nodes, where each node represents an operation call. The set E is the set of all directed transitions (s, t) with $s, t \in N$, where s denotes the source and t the target node. The function $\Delta : E \to \mathbb{N}$ returns the transition count of a given edge. The relation $\Gamma : N \times I$, where I is the set of all call-information and it denotes which node contains which call-information.

Chapter 4

Implementation

The approach introduced in Chapter 3 is realized as a pipe-and-filter system using TeeTime [Wulf et al. 2014] stages. We created a hierarchical stage system using composite stages to divide our tasks into further subtasks. The top level of this hierarchical structure can be seen in Figure 4.1. The highest ordered stage is the `TBehaviorModel`. It takes user sessions, which are represented by `EntryCallSequenceModel`-objects, as input and creates behavior models. Each behavior model represents a user group of the system.

Following our approach starting at Section 3.1 we provide our implementation for entry-calls, which are enriched with call-information in Section 4.1. From these extended entry-calls we create the user groups. We separated this creation of user groups into two phases. In the first phase the incoming entry-call sequence models are prepared for the clustering. This is done by the `TBehaviorModelPreperation`, which is described in detail in Section 4.2. The second phase is the aggregation of the preprocessed user sessions to user groups in the form of behavior models. These behavior models are send to a visualization server. The second phase runs in the `TBehaviorModelAggregation` stage and is described in Section 4.3.

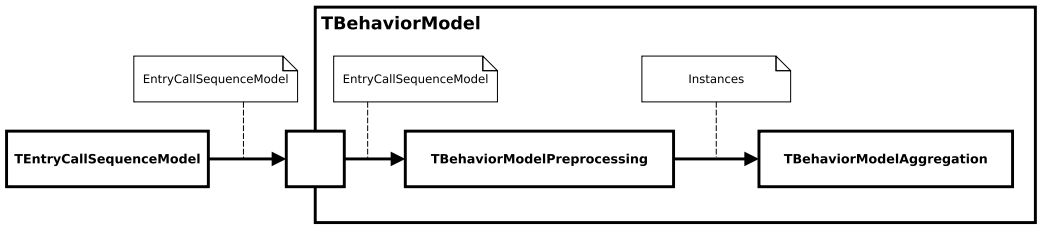


Figure 4.1. Top level stages of the analysis of the approach

Our stage system is integrated in the iObserve analysis. To receive user sessions formatted as `EntryCallSequenceModel` we create another output port in the `TEntrySequenceModel` stage and connect it to our `TBehaviorModel` stage. Every system model created in the analysis is also send to our stage system.

We implemented our approach as a general purpose solution, thus, we provide configurability for special purposes. In Section 4.4 we show the implementation of our configuration object. Finally in Section 4.5.1 we describe the steps for executing our implementation

on a system.

4.1 The Monitoring and Prerequisites

Our first goal defined in Chapter 1 implies that we want to add user specific data to user-behavior and to include it in the clustering. Therefore, we have to define which user specific data we want to consider, while comparing and aggregating user-behavior. We can use globally known data such as contact details or data created by the user while using the software system. The latter is data created at runtime and the input of an operation. To access the data we have to instrument these operations and create records holding the data. The existing monitoring records used to transport user-behavior information, such as `BeforeOperationEvent` and `AfterOperationEvent`, are not sufficient to transport additional behavior information. Therefore, we have to extend these event types to facilitate the collection of additional user data. Instead of an extension, we could also define our own event types. However, this would render existing stages useless, increase the work necessary to implement our own approach and hinder comparison to existing approaches.

We want to be able to store the call-information as a variable list of key-value-pairs in each operation-event. The version of Kieker IRL used in this work is not able to compile records holding arrays. Since we want a set of parameters for each operation, we can either send multiple records or one record containing a list. A simple solution would be to create a new record type containing a single parameter. From the monitoring side of view, this would be the fastest and easiest solution. On the analysis side though, this would cause extra effort, because the parameter records would have to be reconnected to its operation calls. Therefore, we decided to extend the `BeforeOperationEvent` and the `AvterOperationEvent` with a simple field of the type string called `information`. The parameters are stored in the field as a JSON string in the form of key-value pairs. An operation like `public void buy(long itemID)`, which is invoked with `buy(123456789)`, produces a JSON of the form:

```
[{"informationSignature":"itemID","informationCode":"123456789"}]
```

This causes some overhead, but can be replaced with an array more easily, when arrays are available in the Kieker IRL generator.

We called the new created records `ExtendedBeforeOperation` and `ExtendedAfterOperation`. They can be handled by the analysis without problems, since they are inheriting the types `BeforeOperationEvent` and `AvterOperationEvent`, which are already handled by the analysis. To get the operation parameters to the user-behavior analysis we have to extend the `EntryCallEvent` the same way we extended the operation call records. When merging them in the `TEntryCallEvent` stage, we are merging their JSON call-information strings as well. The Kieker IRL definition of the `ExtendedBeforeOperation`, `ExtendedAfterOperationEvent`, and `ExtendedEntryCallEvent` are shown in Listing 4.1.

Listing 4.1. ExtendedAfterOperationEvent defined in Kieker IRL

```
1   @author 'Christoph Dornieden' @since "1.0"
2   entity ExtendedAfterOperationEvent extends AfterOperationEvent{
3       string information
4   }
5
6   @author 'Christoph Dornieden' @since "1.0"
7   entity ExtendedAfterOperationEvent extends AfterOperationEvent{
8       string informations
9   }
10
11  @author "Christoph Dornieden" @since "1.0"
12  entity ExtendedEntryCallEvent extends EntryCallEvent{
13      string informations
14  }
```

The `TEntryCallEvent` stage merges `BeforeOperationEvents` and `AfterOperationEvents` to `EntryCallEvent`-objects. Every incoming `BeforeOperationEvent` is stored in a map. When the corresponding `AfterOperationEvent` arrives, the `BeforeOperationEvent` is retrieved from the map and merged with the `AvterOperationEvent` to an `EntryCallEvent`. For our implementation we have to modify the stage for extended events. Since the extended operation events extends the operation events, the stage can handle them without any modification. Thus, we only need to modify the merging process. When the stage is merging the `BeforeOperationEvent` and `AvterOperationEvent`, we check whether one or both events are extended. For each extended event, we extract the JSON string containing information. If both are extended events we merge the JSONs with string operations to one JSON string. Now we create an `ExtendedEntryCallEvent` instead of an `EntryCallEvent` and add the extracted information to it.

Operation parameters in general can be of any variable type. The problem for the analysis is that our X-Means clustering method from Weka can only handle numeric values of type *double*. Therefore, we have to encode parameters of other types. Since the method of encoding a parameter has impact on the clustering, we can not provide a general encoding concept. The encoding has to be done for every project particularly. In Section 5.2 we have created an encoding for a specific project and explain it in detail.

4.2 The Preprocessing Stages

Before we can aggregate the user-behavior, we have to eliminate unnecessary transitions, presort call-information, and transform the behavior into a format we can use for the clustering. These tasks are all handled in the composite stage `TBehaviorModelPreprocessing`,

which is depicted in Figure 4.2.

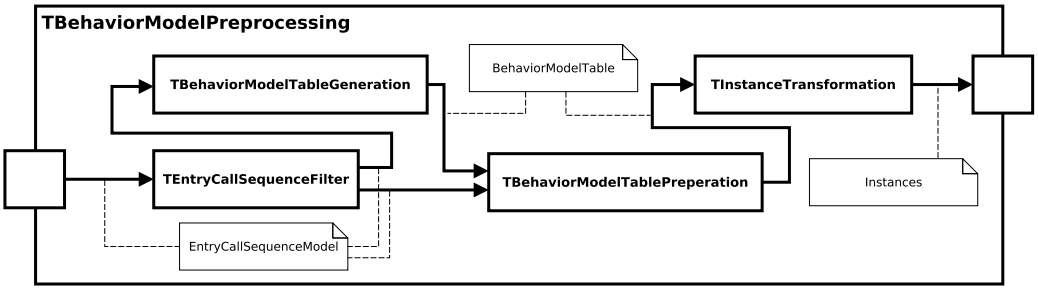


Figure 4.2. Behavior model preprocessing stage

This preprocessing stage contains four stages. Initially all incoming `EntryCallSequenceModels` are filtered by the `TEntryCallFilter`. It filters out all `EntryCallEvents` from the sessions of the `EntryCallSequenceModels` that are not needed for the clustering. Then the `TBehaviorModelTableGeneration` and the `BehaviorModelPreperation` take the filtered entry-call sequence models, and create a behavior model table for each session of the sequence model. The generation stage preprocesses the entry-calls by creating a model table prototype of them. The preparation stage takes this prototype of the entry-call sequence models as input. It transforms each session to a behavior model table. The last stage then transforms the tables to an `Instances` vector for the clustering.

4.2.1 BehaviorModelTable

The `BehaviorModelTable` seen in Figure 4.3 is the implementation of the $n \times n$ matrix from Section 3.3. It contains the matrix as a two dimensional integer array called `transitions`. Further, it holds a map `signatures` and an array `inverseSignatures` corresponding to the functions o and o^{-1} mapping operation signatures to the `transition` matrix and vice versa.

Besides the index of an operation signature, the `signatures` map is also mapping to the signatures on call-information and thereby implementing the function γ. The call-information, $I_{s,U}$, are stored in objects of the type `AggregatedCallInformation`. This class is designed to hold multiple call-information of the same type and is needed when a user accesses the same operation multiple times with different arguments. Each argument is saved as a list of numbers in the `callInformationCodes` field. For the clustering, we can only use a fixed number of parameters, thus, we have to aggregate all codes to a fixed number of codes. We decided to use only one code as representative for each `AggregatedCallInformation`-object. This code is stored in the `representativeCode` field. To find this represented code, a strategy β is necessary. We implemented β as a basic strategy, that returns the call-information with the most occurring value, but it can be exchanged for a custom integration by implementing the `IRepresentativeStrategy` interface.

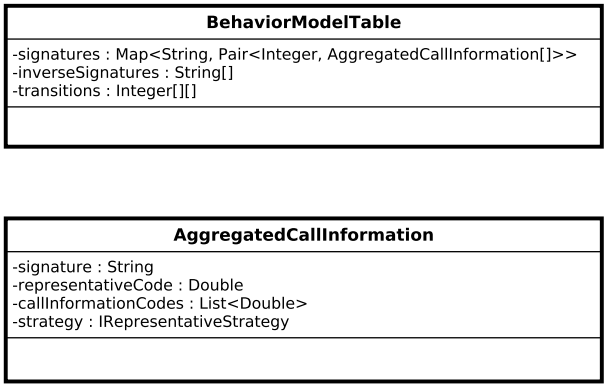


Figure 4.3. Class-diagram of `BehaviorModelTable` and `AggregatedCallInformation`

4.2.2 DynamicBehaviorModelTable

In the `BehaviorModelTable` every array has a fixed size. This is needed to create comparable `Instance` vectors from it. When we get a list of user sessions, we do not know how many different operation calls are in the entry-calls, thus, we do not know the size of the arrays. To overcome this problem we design a dynamic sized behavior model, which can be used as a prototype for fixed-size behavior models called `DynamicBehaviorModelTable`.

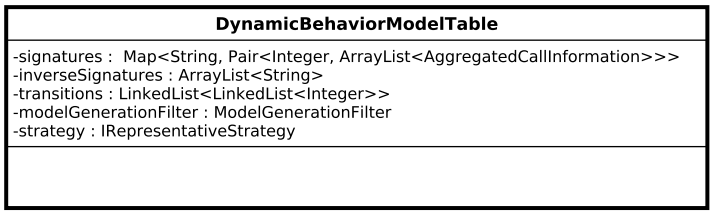


Figure 4.4. Class-diagram of `DynamicBehaviorModelTable`

The members of this class can be seen in Figure 4.5. Since the dynamic behavior model table is transformed to a fixed size behavior model table, the attributes `signatures`, `inverseSignatures`, and `transitions` fulfill the same roles as in the behavior model table. The only difference is that we have lists instead of arrays.

For signatures added to the table, a new entry to the signature map and the inverse signature list is created. Then, the matrix, which holds the transition counts, is extended from the size of $n \times n$ to a $(n+1) \times (n+1)$-matrix. Every new field in the matrix is marked

as an empty transition with the value of -1 until a transition is added for this field. This way we create a sparse matrix. When the generation of the table is finished, we can assume that these fields are not used.

When adding new information, we have to create new aggregated call-information. Since every aggregated call-information needs a representative strategy, the generation stage provides it in the attribute `strategy`.

4.2.3 EntryCallFilterRules

The entry-call filter rules are the implementation of the δ function from Section 3.2. The purpose of this rule set is to verify whether an operation signature is an allowed signature. It contains a boolean and a list of regular expression pattern. If the boolean is *true*, the pattern list is a black list and if it is *false* the list is a white list. A signature verified with this filter is checked against all regular expression patterns. If the filter rules are white list rules, the signature has to match at least one pattern from the rule set. When rules are configured as a black list, the signature is not allowed to match any of the patterns.

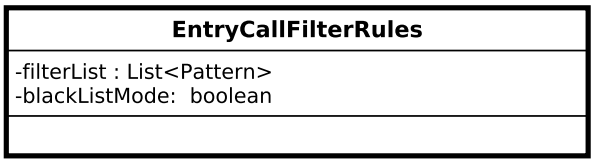


Figure 4.5. Class-diagram of `EntryCallFilterRules`

4.2.4 The TEntryCallFilter Stage

In the `TEntryCallFilter` the filter rules from the `EntryCallFilterRules`are applied. For each incoming `EntryCallSequenceModel` a filtered copy is created and send to the output port. In the filtered copy all `EntryCall`-objects, which signature do not matches the rule set, are removed from the sessions of the input `EntryCallSequenceModel`.

4.2.5 The TBehaviorModelGeneration Stage

The clustering can only handle vectors of the same size and order. Thus, our behavior model tables need to have the same signature index mapping. Therefore, we need an empty predefined table as sample to create new tables of the same kind. To create this sample table we place a table generation stage between the `TEntryCallSequenceModel` stage and the `BehaviorModelPreperation` stage.

The `TBehaviorModelGeneration` stage uses the `DynamicBehaviorModelTable` as inner model to create a prototype behavior model table. On initialization, a new dynamic table is cre-

ated with an IRepresentativeStrategy. The input of the stage are objects of the type TEntryCallSequenceModel. For every entry-call sequence model, every transition of every session is added to the dynamic table.

On termination of the stage, a cleared fixed sized copy of the dynamic table is created and send to the output port. A cleared copy of fixed size is a BehaviorModelTable containing the same signatures as the dynamic table. The difference is that the aggregated call-information only contains a signature and no stored information. In the transition matrix, every value is set to zero, except for empty transitions. For empty transitions, we can choose depending on our scenario whether we want to keep them or not.

4.2.6 The TBehaviorModelPreperation Stage

The TBehaviorModelPreperation receives objects of the type EntryCallSequenceModel and an empty BehaviorModelTable. As described in Section 2.3.3 EntryCallSequenceModel-objects contain lists of EntryCallEvent-objects assigned to sessions which are assigned to users. In our aggregation we want to cluster all user sessions. Therefore we have to extract them from the user sessions. Each session represents a path the user has taken while navigating the system. For our aggregation we want to know which transitions the user took regularly and which he did not.

The TBehaviorModelPreperation stage creates a new behavior model table for each list of EntryCallEvent-objects and adds the call events pairwise to the model. When one EntryCallEvent is an instance of an ExtendedEntryCallEvent, the stored information is added separately to the BehaviorModelTable.

When initialized, the BehaviorModelPreperation stage is storing all incoming entry call sequence models until it gets the prototype table from the generation stage. Then it starts processing first the stored and second the incoming models.

In the TBehaviorModelGeneration stage we use the EntryCallFilterRules to verify if a signature is relevant for the dynamic behavior model table. Since we have a fixed size of the transitions matrix and a fixed number of signatures, we do not need the EntryCallFilterRules. Thus, it is not possible to add new signatures to the model table. If a transition is added with an entry-call signature, which is not in the signatures map, we skip this entry-call after the same concept, we used in Section 3.2.

The incoming models and sequences are processed with the same input port and separated in an if-case via instanceof. That means that the input port is of type Object. In general this is bad style, since we are not maintaining *Separation of Concerns* and disable the type system. The stage implementation used version of the TeeTime Framework only has one execution that is triggered by one input port. Since we want an execution for each element and not only for one type we have to use one input port with two input types.

4.2.7 The TInstanceTransformation Stage

The `TInstanceTransformation` stage transforms a set of `BehaviorModelTable`-objects to a single `Instances`-object. This `Instances` element is needed for the clustering, since we use Weka which is only able to cluster `Instances`. An `Instances`-object is a collection of Weka `Instance`-objects. Each `Instance`-object is an attribute vector, while the `Instances` stores the attribute names for these vectors and the vectors itself. Behavior model tables contain both information in one object, therefore, the information has to be split.

The first behavior model table processed by the `TInstanceTransformation` stage is used to create the `Instances`-object. Every field in the transition matrix and every call-information is transformed into one attribute. To reconstruct our behavior model from these `Instances`, we encode the attribute names. For transitions we use an identifier at the beginning of the attribute name, followed by the source operation name, a unique separator and the target operation. A transition from `operationA` to `operationB` will be encoded as "><*operationA->operationB*". Call information start with another identifier and have a different separator, but the concept is the same. A call-information `infoC` of `operationA` is transformed to "##*operationA~~infoC*". Whereby the value for *infoC* is the representative value of the aggregated call-information of *infoC*. For the encoding we use characters, which are normally not part of a operation signature. When the attribute names for the `Instances`-object are defined and the object is created, incoming behavior model tables can be transformed to `Instance` vectors. The value for each call-information is the representative value of the aggregated call-information.

As we described in Section 3.3.1, for the clustering of the `Instances` it is preferable to have small vectors. Therefore we have to reduce the number of attributes to a minimum. At this point, we have a benefit from using a sparse matrix for the transition values. Every empty transition can be left out from the `Instances`-object. This way we can reduce the number of attributes. Since every behavior model table is derived from the same initial table in the `TBehaviorModelPreperation`, the empty transitions are the same for every table.

4.3 Aggregation and Visualization

The aggregation stage `TBehaviorModelAggregation` is a composite stage directly connected to the output port of the preparation stage. Figure 4.6 shows the aggregation stage with its sub-stages. The first stage is the `TClusteringStage`. It takes the `Instances`-object and aggregates them to clusters. The centroids of these clusters in the form of `Instances` are the output of the clustering. Then, the instances are transformed to `BehaviorModel`-objects in the `TBehaviorModelCreation`. The created behavior model are then send to the `TBehaviorModelVisualization`, which sends the models to the user-behavior visualization component of iObserve.

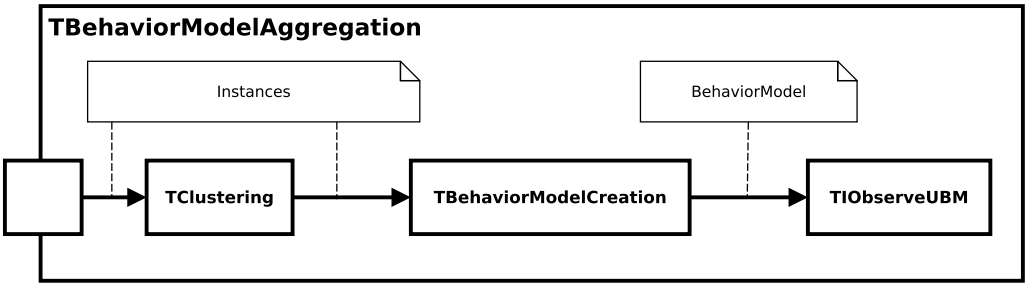


Figure 4.6. Behavior model aggregation stage

4.3.1 BehaviorModel

We implemented the behavior model $\mathscr{M} = (N, E, \Delta, \Gamma)$ described in the approach (Section 3.5), as depicted in Figure 4.7. We realized the graph $\mathscr{M}$ as the `BehaviorModel` class, which contains sets of `EntryCallNode` (N) and `EntryCallEdge`-objects (E). Each edge has an attribute `calls`, which is our implementation of the function Δ. We implemented the function Γ as composition of `CallInformation`-objects in a node.

Each `EntryCallNode` has a signature. This is the identifier of the node and unique in each behavior model. When a node is added to the model with the same signature, both nodes are merged to one. This means that the call-information of both nodes are combined in one node. If two call-information have the same signature, we discard the newest. At the point of the implementation, this scenario is a theoretical case. Since the behavior models are created from the clustering results, it is not possible to have more than one information code per information signature. If the data structure is used in the future, with different use cases, a solution for this problem would be to use `AggregatedCallInformation` instead of `CallInformation`-objects.

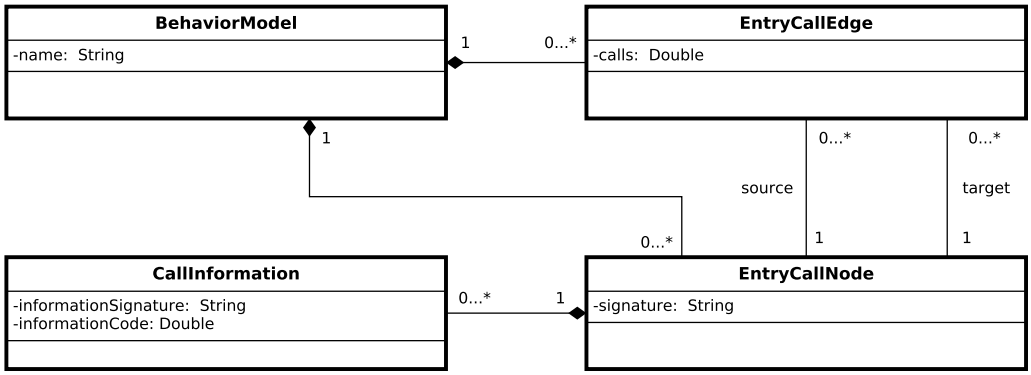


Figure 4.7. Class diagram of the behavior model

4.3.2 The IClustering Interface and its Implementations

We define an interface for clustering algorithms that can be used by our clustering stage. Since there is no clustering fitting perfect for all clustering scenarios, we use an interface instead of one algorithm. A clustering algorithm implementing this interface is required to implement one operation:

```
Optional<ClusteringResults> clusterInstances(Instances instances)
```

This operation takes `Instances` for the clustering and returns a `ClusteringResult` if the clustering was successful. The class `ClusteringResult` was implemented by Peter (2016) and is part of the existing solution in iObserve for user-behavior aggregation. It is a container class for clustering results and provides some meta information like the SSE.

We implement the X-Means clustering in `XMeansClustering`. It implements the interface `IClustering` and uses the `XMeansClusterer` of the Weka framework. The constructor of the class takes the number of the predicted user groups, the variance of this prediction and a distance metric. Since the X-Means clustering depend on the initial k of the K-Means used in the X-Means, we execute the clustering multiple times. We compare the SSE of each solution and return the result with the lowest error count.

4.3.3 The TClustering Stage

The `TClusteringStage` initializes with a clustering implementing the interface `IClustering`. For every `Instances`-object it receives at its input port it executes the clustering method of the interface.

We implement the clustering stage as a general purpose clustering. It can be used in any part of the iObserve framework. We use the TeeTime framework and implement

the TClustering as a stage. It can be connected to any pipe in the iObserve analysis. If a clustering is needed in the future, this stage can be reused. In that case the stage could even be initialized with another clustering algorithm that fits the purpose better.

4.3.4 The TBehaviorModelCreation Stage

The TBehaviorModelCreation takes an instance vector and transforms it into a BehaviorModel. The input instance vector has to be in the same format as provided by the TInstance-Transformation stage. For the transformation to behavior models, the attribute names of the instance are categorized and split into its components. First, an empty BehaviorModel is instantiated. For every transition an EntryCallEdge and two EntryCallNodes are created and added to the behavior model.

If a transition count denotes the number of times the transition is taken we call it an absolute transition count. An absolute transition count can be represented by integer values. Furthermore, a transition count can represent the possibility that a transition is taken. In this case we call it a relative transition count. The relative transition count is a real number between one and zero and has to be represented by a float or double value. The attribute calls of the entry call edges is of type double, therefore it can also be used for relative transitions, but for our approach we want to have absolute transition counts. Therefore, we need positive integers as call count. Vectors created by the clustering can have values of the type double. Since we know that our input for the transition values was positive, we know that the transition values after the clustering are positive too. While single matrices contain only integer values, during the clustering values are aggregated and can become real numbers, e.g., a transition count of 1 and a transition count of 2 will be merged to a transition count of 1.5 in the behavior model. Therefore, we round every call count to the nearest integer before we add the corresponding edge to the model. Is the rounded call value of a edge zero, we do not add the edge to the model. Thereby, a transition with a call count below 0.5 will be discarded.

When the attribute is a call-information, a node is created and the call-information added to it as CallInformation-object . Then the node is added to behavior model. If a node of with the same operation signature already exist in the behavior model, both nodes are merged, by merging all call-informations in one list. Since every field of the Instances vector contains a different call-information, it is not possible to create a node with two CallInformation, that have the same signature.

Each Instance from the Instances is one user group. Therefore, we create a behavior model for each instance vector. If the behavior model is empty, because no edge or node was added, we discard it, otherwise we delegate it to the output.

4.3.5 The ISignatureCreationStrategy Interface and its Implementations

In the visualization, the node is labeled with the full operation signature. Depending on the project name, the nested packages, and the class name these labels can be very long. Long

names create a bigger layout and thereby increase the visual complexity of the behavior model graph. Since we know the packages and class of a operation in the most cases, we do not always need to display the whole operation signature.

The `ISignatureCreationStrategy` interface requires the implementation of the operation `String getSignature(EntryCallNode node)`. This operation takes an entry-call node and returns the displayable name of the node. An object of the interface is needed in the `TBehaviorModelVisualization`, which is sending the behavior models to the visualization server. We created some strategies implementing this interface. The `GetLastXSignatureStrategy` returns a cropped signature with the last x parts of the operation. When it is called with 1 or less it returns only the operation name. When it is called with INT_MAX or a number grater than the number of parts, it returns the full operation signature.

4.3.6 The TBehaviorModelVisualization Stage

We visualize our behavior models with the *User-behavior Model Visualization* for iObserve [Banck 2017] by Daniel Banck, which is described in Section 2.4. Therefore, we create the `TBehaviorModelVisualization` to send our behavior to the visualization backend server.

The internal behavior graph model of the visualization component resembles our behavior model, since both represent a directed graph. The `Application` corresponds to our `BehaviorModel`, the `Page` nodes to our `EntryCallNode`, and the visits to our `EntryCallEdge`. Each page has a list of key-value pairs as property. Since our `CallInformation` class basically is a key-value pair, call-information can be added to pages without great effort.

For each behavior model we process in the `TBehaviorModelVisualization` stage, we create an `Application` with the name of the behavior model at the server. To differentiate between behavior models of different systems, we add a prefix to the name of the model. Then we transform every node of the model to a JSON object matching the `Page`-object. The JSON objects are send to the backend via REST. The visualization backend server assigns its own ids to the elements send to the server. Therefore, we can not use our own ids when accessing the models on the server. Since we can not use our own ids for the nodes, the response for each request is linked to the node send. After that, all edges are transformed to the JSON conforming to a `Visit`-instance of the visualization. The ids from the mapped responses are used to identify source and target nodes of the edges.

4.4 The Behavior Model Configuration

Our Approach is depending on many variables, since we are trying to create a general purpose solution. To be able to store all configuration possibilities, we created a configuration metamodel TBehaviorModel, depicted in Figure 4.7. We use an instance of the configuration class is a parameter of the constructor of the `TBehaviorModel` stage. From there the configuration instance is distributed to the sub-stages.

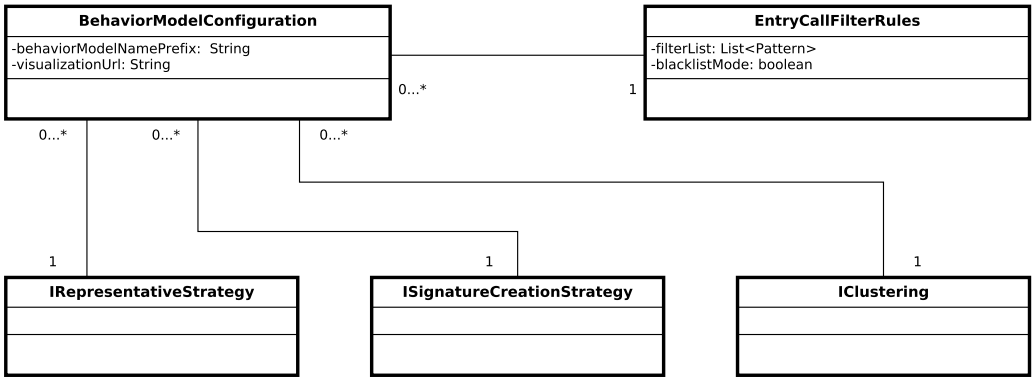


Figure 4.8. Class diagram of the behavior model configuration

The `EntryCallFilterRules` is needed in the preprocessing to filter out non relevant entry-calls. The `IRepresentativeStrategy` is required in the stage, `TBehaviorModelGeneration`. It defines how the representative of a `AggregatedCallInformation` is chosen.

One property of the configuration is the `behaviorModelNamePrefix`. The name prefix is a string referencing the system where the user behaviors are created from. It is used in the `TBehaviorModelVisualization` to extend the name of the behavior model. We need this prefix to separate the models of the different systems in the visualization, since all behavior models are displayed there at once. Two other configuration properties are needed in the `TBehaviorModelVisualization` stage. The first is the URL of the backend server, the behavior models are sent to, and the second is an object implementing the `ISignatureCreationStrategy` interface.

The behavior model configuration has a default constructor providing a basic configuration. The basic configuration contains defaults for all relevant properties and thereby can be used in the `TBehaviorModel` stage without modification. All properties can be changed later. All properties can be seen in the following table:

`behaviorModelNamePrefix`	*BehaviorModel*
`visualizationUrl`	*localhost:8080*
`entryCallFilterRules`	`new EntryCallFilterRules(false).addFilterRule(".*")`
`representativeStrategy`	`new DefaultStrategy()`
`signatureCreationStrategy`	`new GetLastXSignatureStrategy(Integer.MAX_VALUE)`
`clustering`	`new XMeansClustering(1, 1, new ManhattanDistance())`

A notable property are `entryCallFilterRules` and the `clustering`. An `entryCallFilterRules`-instance is set to *whitelist-mode* allowing all operation signatures. The clustering is a X-Means clustering with an output of one cluster. Therefore, the aggregated behavior will be the mean of all behaviors.

4.5 Steps for a custom Integration

The idea of our approach is to be used in a wide spectrum of systems. Therefore, we implemented our approach to be adaptable. To integrate our approach in a custom scenario, configuration steps have to be made. In the following section we describe what configuration has which effect. Note that we assume that a monitoring and a iObserve system is already set up. If this not the case, it has to be done upfront.

4.5.1 Extension of the Monitoring

Our implementation is designed to cluster user sessions containing extended entry-call events. To get extended entry-call events the monitoring has to be modified. In the monitoring call-events have to be created for each operation call. This is done by creating a `BeforeOperationEvent` when the operation is called and an `AfterOperationEvent` when the operation returns. To add call-information to the Events, the extended versions of these events have to be used. The `ExtendedBeforOperationEvent` and the `ExtendedAfterOperationEvent` can either hold a string of call-information. It does not matter which event is used to store call-information and it is not relevant that both operation events are extended. For example, it is possible to create a `BeforeOperationEvent` and a `ExtendedAfterOperationEvent` for the same operation and the analysis will handle them correctly.

For the analysis it is important that the call-information are in the right format. A call-information has two properties. The `informationSignature` and the `informationCode`. The signature is the string identifier of the call-information. The `informationCode` stores the information itself as a number. If the call-information is a string or another data type, it has to be transformed into number. Depending on the information, it can sometimes be more reasonable to split the user information into more than one call-information. We do this in our integration for the JPetstore in Section 5.2.

A list of call-information can be stored as JSON string in the extended operation events. An example string containing a call-information list is provided in Listing 4.2.

Listing 4.2. JSON representation of a call-information

```
1 [{"informationSignature":signatureA,"informationCode":codeA},
2 ...
3 {"informationSignature":signatureZ,"informationCode":codeZ}]
```

It is also possible to use our approach without the extension of the monitoring. The entry-call events created in the analysis entry-call events will not contain call-information, but the aggregation will perform solely on the operation signatures. Then the aggregation is only based on the navigational patterns of the user.

4.5.2 Creation of a Configuration

The second important step for a custom integration is the composition of the configuration object. The `BehaviorModelConfiguration`-instance has to be passed to the `TBehaviorModel` stage. The different properties have to be adapted to system monitored, e.g., to create the `entryCallFilterRules`-instance properly, all important operation signatures have to be known by the operator creating the filter. When we want to do a custom integration we have have to find proper values for the following configuration properties.

EntryCallFilterRules To decide how the entry-call filter rules are designed, we have to know which nodes are relevant in our system. The filter uses Java regular expressions which are applied to the operation signature. We can both create a black or a white list. With the filter rules we can filter out operation calls that are not relevant for the behavior because they are, for example, made by subroutines. Furthermore, we can even allow only calls from a specific sub-system. Thereby, we are able create behavior models for different sub-systems. Important is that the creator of the filter knows which operation calls are monitored on the system and which of them should be relevant for the behavior model creation.

IClustering Essential for a good clustering result is the selection of the right clustering algorithm [Suthar and Oza 2015; Abbas 2008]. Using `IClustering` gives us the opportunity to decide which algorithm we use and how we configure it. We can use any algorithm that is Weka conform, meaning it accepts and returns Weka `Instances`.

IRepresentativeStrategy For each aggregated call-information signature, only one code can be added to the `Instances`-vector for the clustering. Our implementation needs a strategy for handling multiple codes of one call information. The strategy is highly depending on the information coding done in Section 4.5.1. Therefore, it can be necessary to use different strategies for different signatures. For example, we have a website where each page measures the time a user is browsing it an puts this time in a call information. Then, we will have aggregated call-information containing a list of durations. We now have to decide what the representative code for the aggregated call-information should be. If we want to know the overall duration for each user, we create a strategy to sum up all durations. Otherwise, if we want to know the mean duration, we create a strategy finding the mean of all durations.

If use our approach with extended entry-call events, but want to aggregate the behavior on the behavior graph only. We can create a `IRepresentativeStrategy`, which constantly returns the value 0. Thereby, the call-information fields in the input vectors of the clustering do not span another dimension. Thus, the call-information have no effect on the clustering and user behavior is aggregatedIRepresentativeStrategy based on the structure of the behavior graphs.

Chapter 5

Evaluation

In this chapter, we describe the setup and evaluation of our approach and its implementation. In Section 5.1, we compare the behavior models of our approach to the models of David Peter's [Peter 2016] approach. In Section 5.2, we describe how we generate user data of the JPetStore, which we will use for our evaluation. In Section 5.3, we verify whether our implementation is capable of clustering user sessions with user specific data to meaningful user-behavior. And in Section 5.4, we compare the behavior model creation approach by David Peter [Peter 2016] with the results of our approach.

5.1 Setup for the Behavior Model Comparison

In our evaluation we compare the models created by our approach to the models of the approach of David Peter [Peter 2016], since both approaches in are implemented in the iObserve framework

5.1.1 Prerequisites

David Peter's user-behavior model generation is based on the Palladio Component Model (PCM) (see Section 2.3.2). Its behavior models are represented in user usage models. To create these usage models Peter has the use the other models of the framework.

Peter's approach aggregates sequences of `EntryCallEvent` objects to create a usage model. In the usage model, the entry-calls are represented by `EntryLevelSystemCalls`. During the aggregation a Run-time Architecture Correspondence (RAC) is applied to transform the `EntryCallEvents` to `EntryLevelSystemCalls`.

5.1.2 Overview of the Comparison Implementation

The creation of the current behavior models is done in the `TEntryEventSequence` stage of the iObserve analysis. It takes `EntryCallSequences` as input, creates the behavior models and saves them all in a Palladio usage model. To be able to compare these usage models with the behavior models generated by our new approach, we transform the usage model into a representation which can be viewed by out behavior visualization. The model is stored on the file system. For our comparison we transform the usage model to our behavior model

and send it to the visualization. In the visualization we then can compare the two models visually.

We created a composite stage `TBehaviorModelComparison`, which is depicted in Figure 5.1. Its input port is connected to the output port of the `TEntryCallSequence` stage. The composite stage contains four sub-stages. The first stage is the `TBehaviorModel` stage, which is the implementation of our approach. The second is the implementation of the current approach in form of the `TEntryCallEventSequence`. We created a Distributor copying the incoming entry-call sequences by reference and sending one copy to the behavior model stage and one the entry-event sequence stage. We use a copy by reference distributor, because we know that neither of the stages is modifying the data of the sequences relevant to the other stage.

We create the `TUsageModelToBehaviorModel` stage to transform a usage model to behavior models. It is explained in detail in Section 5.1.3. The input of this stage is a usage created by the `TEntryEventSequence` stage. The current implementation of the entry-event sequence stage has no output port. Therefore, we create one and let the usage model be send to the output port, after it is stored to the file system, to be able to receive the usage model the subsequent stage.

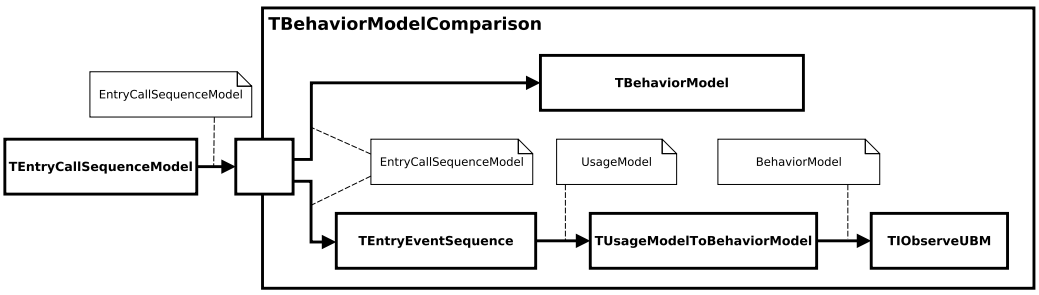


Figure 5.1. Behavior model comparison stage

Finally the behavior models created by the `TUsageModelToBehaviorModel` are send to a instance of the `TBehaviorModelVisualization` stage, which sends them to the visualization. Since the `TBehaviorModel` does the same with its created models, both models are on the visualization server and can be compared.

5.1.3 The TUsageModelToBehaviorModel Stage

In the `TUsageModelToBehaviorModel` stage, we are transforming a `UsageModel` of the PCM to a set of `BehaviorModel` objects. The structure of the `UsageModel` was introduced in Section 2.3.2 and class diagram of it in Figure 2.5. Each behavior of the user group is stored as a `UsageScenario` in the `BehaviorScenario` of the `UsageModel`.

When we are transforming the `UsageScenario` to a behavior model, we create an empty `BehaviorModel` and find the `Start` action of the `UsageScenario`. Then we call the `traverseScenarioBehavior` operation with the usage scenario and the created behavior model. The `traverseScenarioBehavior` operation initializes the traversion of the scenario behavior. It finds its start node, and calls the `traverseAction` operation, which is shown in Listing 5.1, with the behavior model, successor of the start action and an empty `Optional`. [1]

[1] `https://docs.oracle.com/javase/8/docs/api/index.html?java/util/Optional.html`

Listing 5.1. The traverseAction operation of the TUsageModelToBehaviorModel stage

```
1         private Map<EntryCallNode, Double> traverseAction(
2             final BehaviorModel behaviorModel,
3             final AbstractUserAction action,
4             final Optional<Map<EntryCallNode, Double>> optPreviousNodes) {
5
6         if (action instanceof Branch) {
7             final Branch branch = (Branch) action;
8             return this.traverseBranch(behaviorModel, optPreviousNodes, branch);
9         } else  if (action instanceof EntryLevelSystemCall) {
10            final Map<EntryCallNode, Double> endNodes = new HashMap<>();
11            final EntryLevelSystemCall entryLevelSystemCall = (
                  EntryLevelSystemCall) action;
12            final EntryCallNode entryCallNode = this.createEntryCallNode(
                  entryLevelSystemCall);
13            behaviorModel.addNode(entryCallNode);
14            if (optPreviousNodes.isPresent()) {
15                optPreviousNodes.get().keySet().stream()
16                    .map(previousNode -> new EntryCallEdge(
17                                    previousNode,
18                                    entryCallNode,
19                                    optPreviousNodes.get().get(previousNode)))
20                    .forEach(behaviorModel::addEdge);
21            }
22            endNodes.put(entryCallNode, 1.0);
23            return this.traverseAction(behaviorModel, Optional.of(endNodes),
                  action.getSuccessor());
24        } else if (action instanceof Loop) {
25            final Loop loop = (Loop) action;
26            final Map<EntryCallNode, Double> endOfTheLoop = this.traverseLoop(
                  behaviorModel, optPreviousNodes, loop);
27            return this.traverseAction(behaviorModel, Optional.of(endOfTheLoop),
                  action);
28
29        } else if (action instanceof Stop) {
30            return optPreviousNodes.isPresent() ? optPreviousNodes.get() : new
                  HashMap<>();
31        } else { // skip action
32            return this.traverseAction(behaviorModel, optPreviousNodes, action.
                  getSuccessor());
33        }
34    }
```

The traverseAction operation is called with three parameters. The first is the BehaviorModel all nodes and transitions are added to. The second is the current Action the function is called with. The operation processes the action and then calls itself with the successor of this action, thereby the operation is recursive. The last parameter is the optPreviousNodes map. Since we want to create a behavior model, we have to assemble all entry-call nodes in the sequence of actions. When we found a node a, we add it to optPreviousNodes with the value 1. With next node b found by traversing the action sequence, we create a transition from a to b with a probability of one. Since it is possible that one node has multiple incoming nodes, we use a map to store all nodes that have to be connected to the next node found.

In the body of the operation we check the sub-type of the action. If the action is an instance of a Branch, we traverse the each transition of the branch separately by calling the traverseBranch operation. The operation returns a map, where the keys are all end-nodes of the transitions. The value to each key node is the probability of the outgoing transition on the node. Since all nodes from the maps are end-nodes, the probability is always 1.0. In the traverseBranch operation, the inner ScenarioBehavior of each BranchTransition traversed by the traverseScenarioBehavior operation. We call the traverseScenarioBehavior operation with the currentoptPreviousNodes, but change the transition probability of all nodes in it, to the transition probability of the BranchTransition.

EntryLevelSystemCall represent a entry-call and contain a operation signature. If an action is an instance of an EntryLevelSystemCall, we transform it into an EntryCallNode by creating a new node containing the signature of the entry level call (Line 11, Listing 5.1). Now we add the node to the behavior. For all nodes from the optPreviousNodes map we create an edge to the new node and add the edge to the model. Then we create a new map containing the new node mapped to the probability of 1. After that we call traverseAction operation with the successor of the action and the new map.

If an action is an instance of Loop, we call the traverseLoop operation with our behavior model, the map with the previous nodes, and the Loop . In the operation we call traverseScenarioBehavior for the inner behavior of the loop. The traversing of the inner behavior returns the end node of the loop behavior. Then, we create a transition from the end node of the loop to the start node in our behavior model. The operation traverseLoop returns a new optPreviousNodes-instance containing the start node of the loop.

The Stop signalizes the end of the behavior. When the action is an instance of the class Stop, the operation traverseAction returns and thereby breaks the recursion.

When the initial call of the traverseScenarioBehavior operation returns, the behavior model with relative transition counts, passed as property of the operation, contains all operation-calls and transitions of the scenario behavior. We then send it to the output port of the TUsageModelToBehaviorModel stage.

5.1.4 Configuring our Approach for CoCoME

For the comparison of the two behavior model aggregations we use monitored behavior from CoCoME (Section 2.6.2). To execute the current user-behavior generation in the analysis, we have to provide a set of PCM models. These models,e.g., the *repository-model*, have been all created before and are up to date. Additionally a set of monitoring data is already provided. In the following we describe our configuration instance and the reasons for the usage of each property.

entryCallFilterRules The current behavior model generation uses the internal models to sort necessary and unnecessary operations. The resulting behavior model is only containing relevant operations. Instead of the model to analyze the operations from the monitoring, we use our `EntryCallFilterRules` tho filter the operation signatures. In our evaluation we want to make a comparison based on designed workload for the user *Cashier* of CoCoME. The operations relevant for the behavior of the *Cashier* are:

{...}.cashdeskservice.CashDesk.startSale(...)

{...}.cashdeskservice.barcodescanner.BarcodeScanner.sendProductBarcode(...)

{...}.cashdeskservice.CashDesk.finishSale(...)

{...}.cashdeskservice.CashDesk.selectPaymentMode(...)

Therefore, we use the `EntryCallFilterRules` in the *whitelist-mode* with the rule:

$$.*(cashdeskservice)\backslash.(\backslash w*\backslash.)*\backslash w*\backslash(.*$$

signatureCreationStrategy For our `signatureCreationStrategy` we use an instance of the class `GetLastXSignatureStrategy` introduced in Section 4.3.5. The relevant operations for the *Cashier* are listed in Section 5.1.4. We can see, that the last part of the signatures is unique. Thus, we can choose $x = 1$ as input parameter for the `GetLastXSignatureStrategy` strategy.

representativeStrategy The CoCoME workload is used for the comparison of user-behavior, that does not contain call-information. Hence, the entry-calls do not contain call-information and the `representativeStrategy` is never used. Therefore, we can set an arbitrary strategy as `signatureCreationStrategy`. We choose the `DefaultStrategy`, which is always returning the first information-code of the aggregated call-information.

clustering For the clustering, the approach of David Peter [Peter 2016] uses the X-Means algorithm in combination with the Manhattan distance from the Weka tool suite. Therefore, we use our `XMeansClustering` with the Manhattan distance metric. The parameters `expectedUserGroups` and `varianceOfUserGroups` is set in the Section 5.1.5, since it is depending on the designed user groups.

5.1.5 Design of the Workload for the Cashier of the CoCoME instance

For the comparison of user sessions without user specific data, we use user sessions from an CoCoME instance and the configuration of Section 5.1. Therefore, we simulate different customers at the cash desk of a store. We model these customer behavior by creating a three different cashing processes with Selenium. The resulting records are the input for our analysis configured for the comparison. The final clustered user groups of both approaches are then evaluated with the metrics from our GQM evaluation plan.

After login, the cashier performs a loop of four actions, which are depicted in Figure 5.2. The first action is the *startSale* operation which starts a new checkout process, meaning the cashier serves a new customer with a new set of items. Then he starts scanning products with the barcode scanner. For each item scanned the action *scanProductBarcode* is called. After all items are scanned the action *finishSale* is called. Then a payment method is selected via *selectPaymentMode*. When the payment method is selected, the customer pays his items and the purchase is finished.

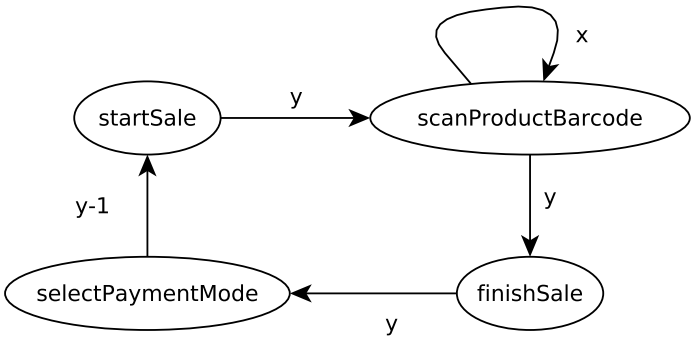


Figure 5.2. General behavior model of the CoCoME cashier with $x, y \in \mathbb{N}$

In this behavior, we can see two loops. One loop is the whole checkout process. The number of its loops is described with the number $y \in \mathbb{N}$. Since a checkout process begins at *start* and ends with *checkout*, the transition between these two has a count of $y - 1$. The other loop represents the scanning of item barcodes. If x items are scanned, the *scanBarcode* action is called x times. Note that we only consider completed checkouts in Figure 5.2. If a checkout is aborted, e.g., a system failure, the behavior does not represent the scanning of item barcodes. To design a scenatio for the cashier, we execute the scenario for a pair $(x, y) \in \mathbb{N} \times \mathbb{N}$.

We designed three different behavior scenarios. In the first we have one customer buying eight items, resulting in $x = 8, y = 1$. Then we have $x = 1, y = 8$ by having eight customers with one item each. Finally we create behavior for four customers with four items each, which entails $x = 4, y = 4$.

We create these user-behaviors with the Selenium framework [Selenium]. This is done by recording one checkout process with the Selenium IDE and converting the recorded script to a Java file. In the file we add the two loops at the beginning of the checkout and at the barcode scanner. To generate the test data we execute this test script on a CoCoME instance 20 times for each behavior. In addtition, we used monitoring data and tooling from David Peters' thesis [Peter 2016] for a proper comparison.

5.2 Setup for the JPetstore Evaluation

We use generated user data of the JPetstore[2] in our evaluation. Therefore, we implement an integration of JPetstore for our approach. To setup the JPetstore for our approach we followed the steps from Section 4.5. First, we extend a instrumented version of the store to create events containing call-information described in Section 5.2.1. Second, we created a configuration for the `TBehaviorModel` stage discussed in Section 5.2.2. Third, in Section 5.2.3 we design the workload, that is created on the JPetstore and used in the evaluation.

5.2.1 Prepare the Monitoring

To evaluate our approach, we use a instrumented version of the JPetstore[3] and analyze the monitored data to create our behavior model. The instrumented version of the JPetstore is provided by the Kieker project. The `SessionAndTraceRegistrationFilter` is added to the `web.xml` of the JPetstore jetty container. It creates events for every page call of the web site and sets the URL as operation signature of the event.

We extended the original iObserve/Kieker SessionAndTraceRegistrationFilter to collect our own version of the `SessionAndTraceRegistrationFilter` to determine and to add information to the call events. Since the filter, filters the URLs we have to get our call-information out of the URL. We recall from the Section 2.6.1, that the JPetstore controller are stripes actions. Each URL call is directed to an action, and the action executes the corresponding operation. If the operation requires parameters, they are added as query to the URL. Thus, we can treat these URL like operation-calls. When we have the calls like:

```
{host}/jpetstore/actions/Catalog.action

{host}/jpetstore/actions/Catalog.action?viewCategory=&categoryId=FISH
```

We notice that, if a operation of the action is called, it is the first parameter of the query. The second query parameter, if there is one, is a parameter of the operation. We want to generate the following operation-calls from them:

```
jpetstore.actions.Catalog.index())
```

[2] `https://github.com/mybatis/jpetstore-6`

[3] `https://github.com/kieker-monitoring/kieker/tree/stable/kieker-examples/JavaEEServletContainerExample/jetty`

```
jpetstore.actions.Catalog.viewCategory()
```

The query parameters of the operation `viewCategory()` are then added as additional information to the call events.

The analysis expects for every operation-call an `BeforeOperationEvent` created on entering the operation an a `AfterOperationEvent` created on the return of the operation. Since we only get the URL calls, we can only register the before-operation events. We, therefore, have to assume, that the operation returned and create an after-operation event for the analysis.

The transformation of a URL is done by several string operations. When the filter gets an URL, the first step is to replace all slashes with dots. Then we check whether the URL has query parameters by searching for a questionmark. If we found none, we just have to replace the last occurrence of the word "action" with "index" and add "()" at the end. Else we extract the operation and its parameters from the query. We do this by splitting the query in its components. This can be done by splitting the query string at the "&"s. We now replace the last occurrence of word "action" with the found first parameter and add "()" at the end. The other query parameter is split into its key and its value and saved as call-information.

In the clustering we can only handle numerical data. Therefore, we have to encode all values to numbers. The encoding is a challenging task, because it is highly relevant for the clustering. In the JPetstore, almost every call information of an operation is describing one element of a group. For example, the operation-call for viewing a certain category of the store contains a attribute called *CATEGORY*. Each category has a different String. When we want to create a call-information for this operation-call, we have to decide how categories should be clustered. We could create one attribute called *Category* and each category has a different integer value. This could look like: *FISH=100,CATS=200,DOGS=300*. When we cluster two vectors and one has a category value of 100 and the other a value of 300, the merge of these two would be 200. We aggregated category *FISH* and *DOGS* and the result is *CATS*. Thus, we create a call-information for each category separately. If the category is *FISH*, we add the call-information *FISH : 1* to the event call. Now the clustering will only merge numbers of one category instead of different category codes. Thereby we increase the dimensional complexity of input vectors for the clustering. However, this will avoid the described issue and may lead to better clustering results.

In addition to the categories, the user can view products and items. A product is a specified category for a set of items, whereby an item is a specific animal. For example, there is the product type *FI-SW-01* of the category *FISH*. *FI-SW-01* is the id for the product type *angler fish*. It contains two items *EST-1* and *EST-2*, which are small and large angler fishes.

We can use either an `ExtendedBeforeOperationEvent` or an `ExtendedAfterOperationEvent` to store call-information. Since both will be merged in the analysis, it is not relevant, which object is storing the data. Therefore, we can choose the event arbitrarily. We pick the `ExtendedAfterOperationEvent`.

Not all URL calls are directed to an action, sometimes images or CSS elements are loaded via GET-Request. We transform them to our operation format by replacing the slashes with dots. Then we create the two entry call events. Since we do not need them for the analysis, we could discard all image and css calls, but we want to demonstrate the use of the `EntryCallFilterRules` in the analysis.

5.2.2 Creating a Behavior Model Configuration

entryCallFilterRulesr For our behavior models, we use only the actions directly called by the user. From Section 5.2, we know that we we have actions of matching the following regular expression:

$$jpetstore\backslash.actions.*$$

Thus, we can set our rules to *whitelist-mode* and add the regular expression to it. Thereby, we allow user actions only and exclude all image and css requests.

representativeStrategy The call-information, which are created while browsing the JPetstore, are information about the categories, products, and items the user is viewing or buying. As described in Section 5.2.1, each of the call-information contains the identifier of the category, product, or item and the number it was viewed/bought as call-information code. Since we create only one call-information per view/buy, the code is always 1. We design our `IRepresentativeStrategy` to sum up all codes. Thereby, the representative code of the aggregated call-information is the number of the contained call-information codes.

signatureCreationStrategy For our `signatureCreationStrategy` we use an instance of the class `GetLastXSignatureStrategy`, introduced in Section 4.3.5. To find an appropriate $x \in \mathbb{N}$ as input for the constructor of the strategy, we have to find the minimal x so that the created signature for the visualization is unique. The operation signatures of the entry-calls from the JPetstore are created by the transformation of the URL of the requested page. In Section 5.2.1, we explained that every direct request on an action is stored as call of an *index()*-function. Thereby, we cannot choose $x = 1$, since multiple operation signatures end with *index()*. Furthermore we designed our `EntryCallFilterRules` to allow only signatures matching the pattern "$jpetstore\backslash.actions.*$". Hence, we do not need to choose $x > 2$, because every signature string will start with *jpetstore.actions.*. Thereby, we choose the `GetLastXSignatureStrategy` with the value $x = 2$ for our `signatureCreationStrategy`.

clustering For the clustering we use the X-Means algorithm with the Manhattan metric as distance metric. The reason we use this combination is explained in Section 3.4. The number of expected user groups and the variance is set later, because it is different some executions of the evaluation.

5.2.3 Design of the Workload for the JPetstore

For our evaluation, we design five different user types for JPetStore. All users start on the main page of the store, which is the `Catalog.index` call.

In Figure 5.3, we depict all possible transitions our user types take in the system. Thereby, the behavior of each user type is a subgraph of it.

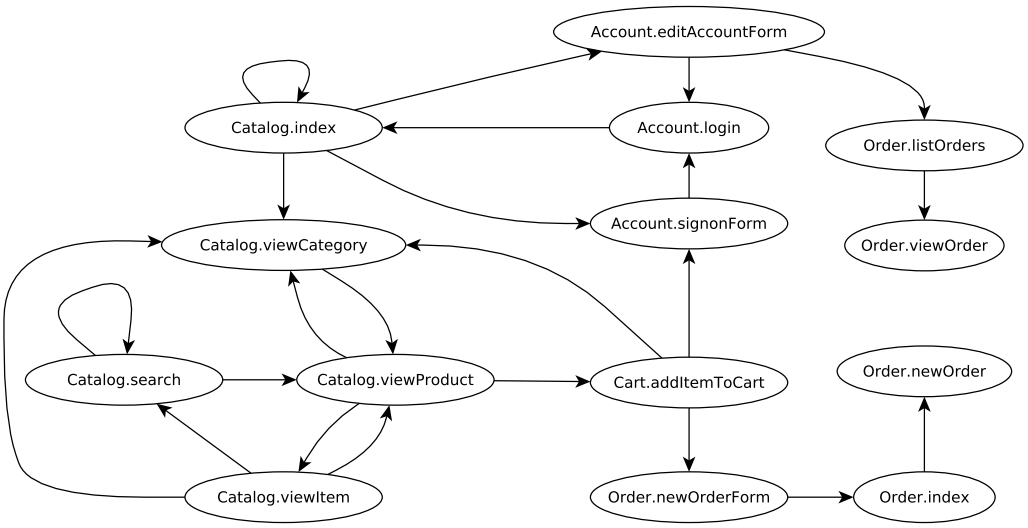


Figure 5.3. Overview over all transitions the created user types take in the system

The call-information are not depicted in the Figure 5.3. Nevertheless the user types produce such, e.g. by selecting categories, products, and items in the store. The categories of the store are *FISH*, *DOGS*, *REPTILES*, *CATS*, and *BIRDS*. The products and items that belong to these categories, have cryptic names that will not be explained, because we will focus on the categories in our user type creation. Every time we name a product or item, it is always obvious to which category it belongs to.

Clustering becomes harder the more similar the user types behave. Thus, we designed all user types to be unique, but to share sub-graphs, transitions, or call-information with other user types. In the following, the user types are presented:

Fish-Lover The user *fish-lover* enters the store, goes to the category *FISH* and adds 8 fishes of the type *FI-SW-01* to the cart. He first signs into the shop, goes to the Category *FISH* and adds one additional fish of the type *FI-FW-01* to the cart. He then proceeds to the checkout, creates a new order, and opens the order view.

Cat-Lover The *cat-lover* performs the same actions as the *fish-lover*, but instead of buying items of the category *FISH*, he is buying products of type *FL-DSH-01* in the category *CATS*. We designed the *cat-lover* to evaluate if our clustering can differentiate between behaviors that are structurally similar but have different call information on the entry calls.

Single Fish Buyer The *single fish buyer* selects the category *FISH*, navigates to the view for product*FI-SW-01*, and adds the first item listed to the cart. He then proceeds to checkout by first logging in and then ordering the item from his cart. The *single fish buyer* uses the same transitions as the *Cat Lover* and the *Fish Lover*, but with different transition counts.

Single Reptile Buyer The *single reptile buyer* acts like the *single fish buyer*, but goes to the category *REPTILE* and orders an item of the product *RP-SN-01*. Both *single buyer* users buy exactly one item as per definition. Therefore the distance between both is much smaller than the distance between the *fish-lover* and the *cat-lover*. We expect that both *single buyer* user types will be merged, if the clustering has low input range.

Browsing User The *browsing user* visits different categories, products, and items, but never buys anything. He therefore does not take the path of the ordering process. He begins by navigating to the *REPTILES* category and then the product view for item *RP-SN-01*. After that he continues with browsing different products and items from the from the category *BIRDS*. After visiting the category four times, he uses the search bar to query for fishes. He finds one fish product and its item and then terminates the session. We designed this user to include sub-graphs in common with the user types we already introduced, but is missing some transitions of these types, as well as having unique sub-graphs, e.g. the search.

New Customer Every user type we introduced is using the same user account to log into the store. The *new customer* enters the store and registers as new customer. Then he logs into the store with his new account and buys an item from the product *RP-SN-01*, which belongs to the category *REPTILES*. This user is sharing sub-graphs and call-information with the *single reptile buyer*.

Account Manager The `account manager` is changing his contact information in the account management after login. After that he is inspecting one of his prior orders. The `account manager` is the most different user we designed. The only transition it shares with other users is the login transition. We expect that this user type is the easiest to find by the clustering.

We create a Selenium [Selenium] script to simulate each user type and execute all script sequentially. The clustering only finds groups of users if multiple user share a behavior.

Therefore, we run each user script 20 times. After each run of a script, we reset the browser used for the script execution. This creates a new session for each script execution.

5.3 Finding Predesigned User Groups in JPetstore Workload Data

In the first part of our evaluation, we want to verify that the implementation of our approach is capable of clustering user sessions with user specific data into meaningful user-behavior models. To achieve this, we design workload for the JPetstore which contains different behaviors. Our goal is to detect these behaviors solely based on the monitoring data. Furthermore, we want to verify the reproducibility of the results we get from one clustering execution run. We test this by executing our analysis on the same workload several times. Furthermore, we run the evaluation for different overlapping input ranges of the X-Means clustering to verify that the clustering is not depending on the given input range.

Following the GQM framework, the *object of study* is the user-behavior that is created by the implementation of our approach. We define our goal with the *purpose* of covering the predesigned user types in the *context* of the JPetstore. For that our *focus* lies on the structural equality of the user types and the user-behavior models. The *stakeholders* of this goal are the developer of iObserve, i.e. we.

Goal Creation of user-behavior models that match with the predesigned user types of the JPetstore with the focus on structural similarity from the perspective of an iObserve operator/developer.

5.3.1 Questions: To what extend do the behavior models match the predesigned user types?

As stated in our goal, we want to match the predesigned user types with the behavior models we generated in our clustering. Therefore, this questions aims to discover how suitable our behavior models are to represent the predesigned user types.

Q1: At which input range of the X-Means clustering do we get the best match of behaviors? The result of clustering algorithms depends, among other things, on the configuration parameters. For the X-Means algorithm we can define the range of expected clusters. The algorithm will then find a fitting set of clusters within this range. We want to verify for our X-Means implementation always returns the best result for the given range.

Q2: To what extend do the behavior models of the best clustering result match the predesigned user types? We expect behavior-models of different structure and quality as

we execute the clustering multiple times within different input parameters. To answer our question Q1 we consequently have to ask to what extend the best models of the clustering iterations match the user types.

5.3.2 Metrics

M1: Coverage With this metric, we check if one behavior graph is a part of another behavior graph. Thus we check whether one behavior graph is the subgraph of another. A graph $\mathcal{A}$ is a subgraph of a graph $\mathcal{B}$, if its nodes and edges are a subset of the nodes and edges of $\mathcal{B}$. This means that a behavior model $\mathcal{M}$ is part of a behavior model $\mathcal{N}$, if all pages and visits of $\mathcal{M}$ are in $\mathcal{N}$ and for all visits in $\mathcal{M}$ applies that the call count of the visits from $\mathcal{M}$ are lower or equal to the call counts of the visits from $\mathcal{N}$.

M2: Similarity Ratio The similarity ratio represents the number of differences and similarities of two behavior graphs and the call-information on each node. To count the differences between two behavior models, we compare the sets of pages and visits from both behavior models. Each different node, different edge and call count is summed up. Thereby, we get the number of differences between two models. Vice versa, we get the similarities of the models. The result is the quotient of the differences and the similarities. If one sum is zero, we say that the models are either equal or unequal.

M3: Sum of Squared Errors (SSE) The SSE sums up the distances between all vectors to its clusters centroid. The higher the SSE, the worse are the clusters. The user types we designed have no variation in its structure. Hence, all created vectors are equal. If our clustering finds all user types, the centroids would lie exactly on these vectors, having a SSE of 0. To measure how close we are to a total match we use the SSE.

5.3.3 Results of the Clustering

We run the clustering of the designed workload with a $varianceOfUserGroups = 3$ and with an increasing number of user groups from 2 to 9 as a parameters for the X-Means algorithm. For each number of expected user groups we run our analysis five times. The number of clusters we get for each number of user groups are denoted in Table 5.1 with their corresponding SSE.

Table 5.1. Clustering result for our designed user types

Expected User Groups	Range	Behavior Models	Avg. SSE	SSE variance
2	[1...5]	2	ca. 620	0
3	[1...6]	2	ca. 620	0
4	[1...7]	2	ca. 614	0
5	[2...8]	4	ca. 311	
6	[3...9]	6	ca. 73	1
7	[4...10]	4	ca. 225	0
8	[5...11]	5	ca. 225	0
9	[6...12]	6	ca. 138	9

The results from the analysis are stable for each number of expected user groups. Each execution produces the same number of behavior models with a low variance for the SSEs. In the Table 5.1 we can see that the number of resulting clusters varies. The best SSE score is achieved with 6 expected user groups as input parameter for the algorithm. The other scores are at least 90 percent higher.

If six behaviors is the best we can achieve with the X-Means clustering, we should have found six behavior models for all expected user group values in the range of three. This should hold true, since the X-Means clustering algorithm is designed to find the optimum number of clusters within a range. Thus, we can assume that the X-Means algorithm is not fit for the task of clustering user behavior in higher dimensions.

By reviewing Table 5.1 we can answer the question for the best input rage (Q1.1) using the SSE (M3). The best results are achieved in the range [3...9].

To answer Q1.2 we examine each created behavior model and determine which user types are part of it. The first created behavior-model is depicted in Figure 5.4. The user views the category *FISH* 9 times, then continues to view two different fish products, and adds 9 of them into his cart. At some point in this process of adding items to the cart, he logs in and navigates back to the category via the index page. Finally, he orders all items from the cart. If we apply the coverage metric M1, we see that the model covers the user type *fish lover*. With the similarity ratio metric M2 we can detect that the structure of the model is not only covered, but also similar to the user type *fish lover*.

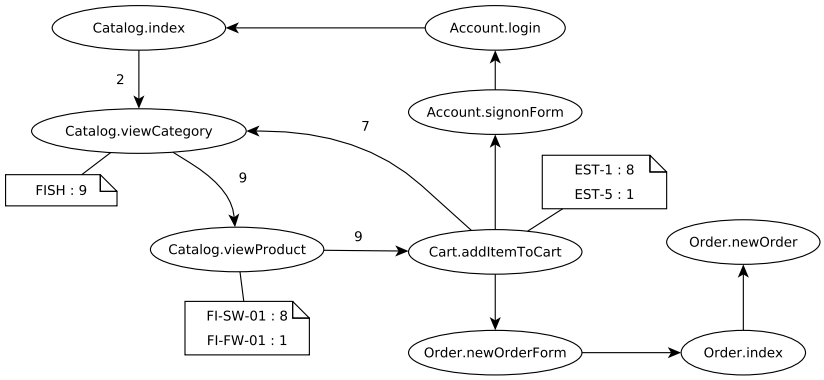


Figure 5.4. Generated behavior-model 1 of the JPetstore workload clustering

The second behavior is shown in Figure 5.5. It looks like the first behavior-model with the difference that the visited items and products are from the category *CATS*. From the definition of the user types, it follows that this matches with the *cat lover* user type.

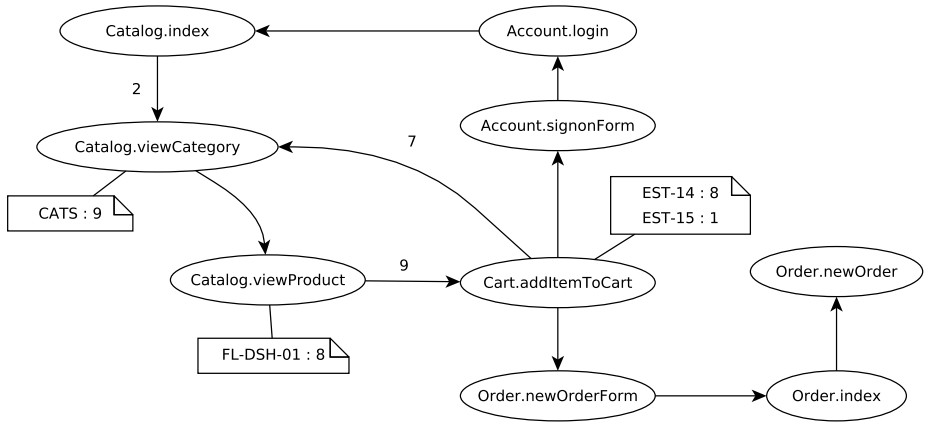


Figure 5.5. Generated behavior-model 2 of the JPetstore workload clustering

Figure 5.6 depicts the third behavior-model, where the *Catalog.viewCategory* is accessed five times. One time for the category *REPTILES* and four times for the category *Birds*. Then the store is navigated further, followed by opening different products and items from different categories. If we apply the coverage metric M1 on this behavior comparing with the *browsing user*, we get a structural coverage. The similarity ratio M2 confirms that the call-information are also the same.

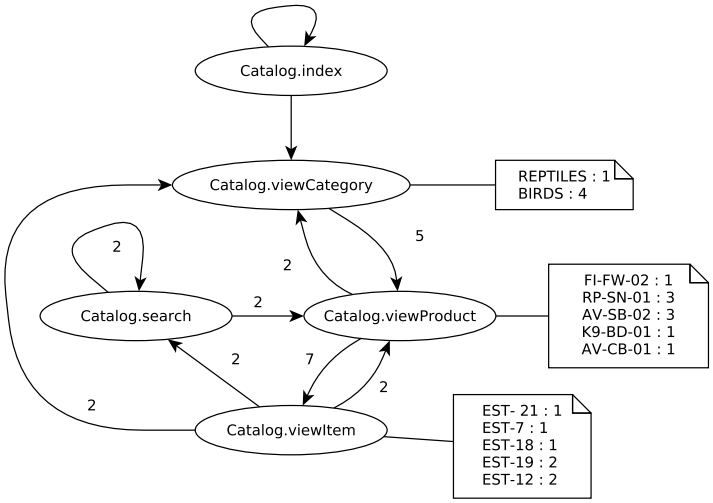


Figure 5.6. Generated behavior-model 3 of the JPetstore workload clustering

The fourth found behavior is depicted in Figure 5.7. In contrast to the other behaviors it only uses single transitions and mainly uses pages managed by the *Account* and the *Order* controller. The user logs in, edits his account information, and reviews an order. This matches the user type *account manager*, as the similarity ratio metric M2 indicates.

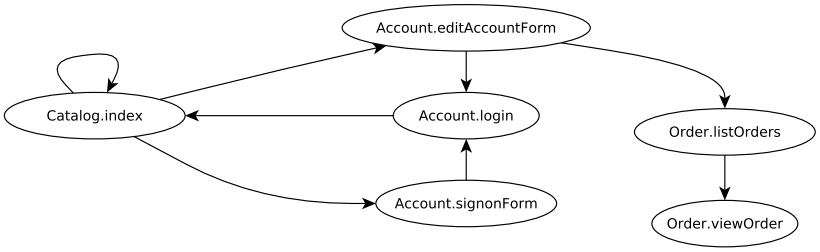


Figure 5.7. Generated behavior-model 4 of the JPetstore workload clustering

In the Figure 5.8, the fifth generated user behavior is shown. We see that the structure almost matches the behavior of the *cat lover* and *fish lover* user type, but the login sequence is at the beginning of the navigation path. The user signs in, opens the category *FISH*, inspects a product, buys an item, and checks out the order. This behavior matches with the *single cat buyer*.

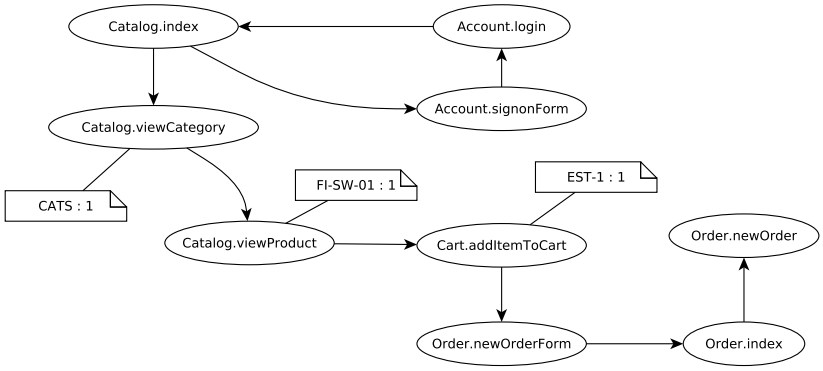


Figure 5.8. Generated behavior-model 5 of the JPetstore workload clustering

The last behavior-model, depicted in Figure 5.8, includes two user types: the *new customer* and the *single reptile buyer*. Both buy the same item from the store without taking any detours in the store. Thereby, the subgraph starting at *Catalog.viewCategory* matches for both user types using the coverage metric M1. The difference between the user types lies in the login process. The *single reptile buyer* signs in with his credentials and the *new customer* creates a new account, after he clicked sign in and before he can log in. Thereby, the connections *Account.signOnForm to Account.newAccountForm*, *Account.signOnForm to Account.login*, and *Account.newAccountForm to Account.login* only have call values of 0.5 and are rounded down to zero. In summary, both user types have a high similarity to the behavior model, but do not match the behavior in Figure 5.8 exactly.

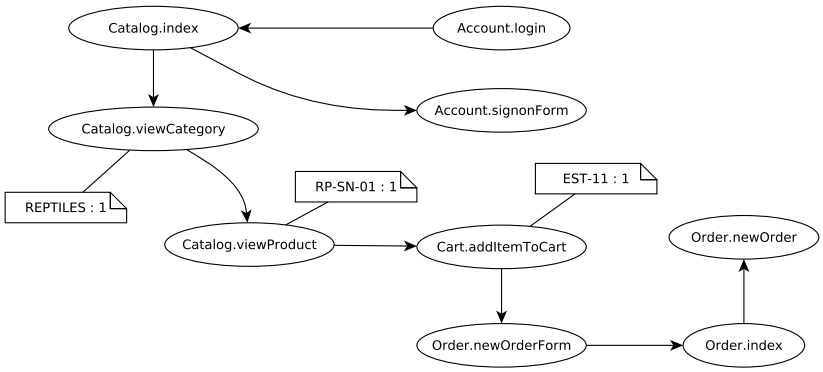


Figure 5.9. Generated behavior-model 6 of the JPetstore workload clustering

5.3.4 Summary

Our goal was to be able to detect all designed user types with our analysis. However, this goal was not achieved completely. In Section 5.3.1, we identified that the resulting behavior models are highly similar to the defined user types. The two user types, that where merged together, are very close to each other. Therefore, the matching of the best results from the clustering is acceptable.

If we look at all results of the execution, we observe that the clustering results are highly varying in their quality. Despite using X-Means with a input range of desired clusters, we have to guess the number of clusters in advance, execute the algorithm for each guess and take the best solution. If X-Means fails to find the best cluster in a range for designed data we can assume that it also fails for real world data, where the user behavior is varying and we have outliers, which can not be clustered. Thereby, X-Means is not a feasible algorithm for this scenario.

5.4 Comparison of Our Approach With the Existing User-Behavior Without User Specific Data

In this part of the evaluation we compare the behavior model creation approach by David Peter [Peter 2016] with the results of our approach. We do this by using the Goal-Question-Metric framework, which is explained in Section 2.6.4. With the evaluation we want to show, that our implementation produces results as good as the results of Peters implementation for behavior creation without user specific data. David Peter showed in his thesis, that his behavior models represent monitored workload characteristics precisely. Since our and Peters approach based on the same system, we do not expect to achieve significantly better results as long as we reduce our approach to cluster user-behavior only.

For this comparison we use an instance of the CoCoME to create the monitoring records and the setup from Section 5.1 to generate user-behavior models from both implementations. The behavior models are visualized through the methods described in Section 2.4. The evaluation is mainly done by a visual inspection on the visualization. We therefore use the terminology of the visualization to describe the behavior models.

The first step for the usage of the GQM is to define one or more goals for this evaluation. In this section we evaluate the implementation of our approach. Therefore, the *object of study* is the implementation of our approach and the *purpose* is to evaluate by comparison. The *focus* lies on structure of the models, because we will compare the created models visually. The *stakeholder* is implied by the iObserve context. Therefore, it is the operator of a software system and subsequently also the developer who can use the models at design time.

Goal The goal is to compare the behavior models created by our approach with Peters behavior models, while ignoring user specific data, with the focus on the structure of

the models from the perspective of an iObserve operator/developer in the context of the CoCoME.

5.4.1 Question Q1: How similar are the results?

We want to show that our approach yields comparable or better results regarding user-behavior clustering than the approach from David Peters. With the first question we want to check, how similar our results are to the other. If we have a high similarity, we can assume that comparable results to Peters approach, if not better. To answer this question, we divided our question into three sub-questions.

Q1.1: How many models have a high similarity? With the first sub-question, we want to find all models that are almost equal or equal from both results. If all models have a high similarity, the results have high similarity.

Q1.2: How different are the most equal and most different models? This sub-question asks for the difference of the edge cases from the results. If there is a high gab between these differences, it is an indicator for a low similarity, even if many models have a high similarity to each other.

5.4.2 Question 2: Is one result more relevant than the other?

With the second question we want to find out if one result is better than the other. If a model has a hight significance, it is more relevant than another. Significance, measures how well the user groups represented by the system can be outlined.

Q2.1: Are behavior models sub-models of other behavior models? A model is more significant, if it describes a characteristic of a of another model. Sub-models of a behavior models, are special characteristics of a behavior model. Therefore, they are more significant. With this question, we try to find out if some models from one result are more significant, than models of the other result.

Q2.2: Does a merged model has a high similarity to another model? A merged model, is a model that is merged together from different models. If two or more highly different models $m_0 \dots m_n$can be merged to another exiting model e, these sub-models $m_0 \dots m_n$ present each a characteristic of the model e. Thereby, they are more significant. This questions tries to find similar models as Q2.1. Q2.1 asks for real sub-models, but sometimes a merge of models can be similar to another model, even if the models of the merge are no sub-models.

5.4.3 Metrics

M1: Sub-Model With this metric, we check if one behavior model is a part of another behavior model. Thus, we check one behavior graph is the subgraph of another behavior model graph. A graph $\mathcal{A}$is a subgraph of a graph $\mathcal{B}$, if its nodes and edges are a subset of the nodes and edges of $\mathcal{B}$. Thus, a behavior model $\mathcal{M}$ is part of a behavior model $\mathcal{N}$, if all pages and visits of $\mathcal{M}$ are in $\mathcal{N}$ and for all visits in $\mathcal{M}$ applies, that the call count of the visits from $\mathcal{M}$ are lower or equal to the call counts of the visits from $\mathcal{N}$.

M2: Similarity Ratio The similarity ration represents the number of differences and similarities of two behavior models. To count the differences between two behavior models, we compare the sets of pages and visits from both behavior models. Each different node, different edge and call count is summed up. Thereby, we get the number of differences between two models. Vice versa, we get the similarities of the models. The result is the quotient of the differences and the similarities. If one sum is zero, we say that the models are either equal or unequal.

5.4.4 Results of the Clustering

We executed our implementation and the approach of David Peter by using the setup from Section 5.1. The version of David Peters implementation we are using is the commit with the hash identifier of `c39addff830a2f06949dfc17a9792a5696689f43` from github.com[4]. The clustering result of David Peter are two clusters. One of these clusters is depicted in Figure 5.10. We can see that version we are using has some errors in the merging process, since every operation is displayed multiple times. At the point in time we recognized this error, we were not able to fix it before the deadline of the thesis. The evaluation can therefore not be fully concluded due to external dependencies.

[4] `https://github.com/research-iobserve/iobserve-analysis`

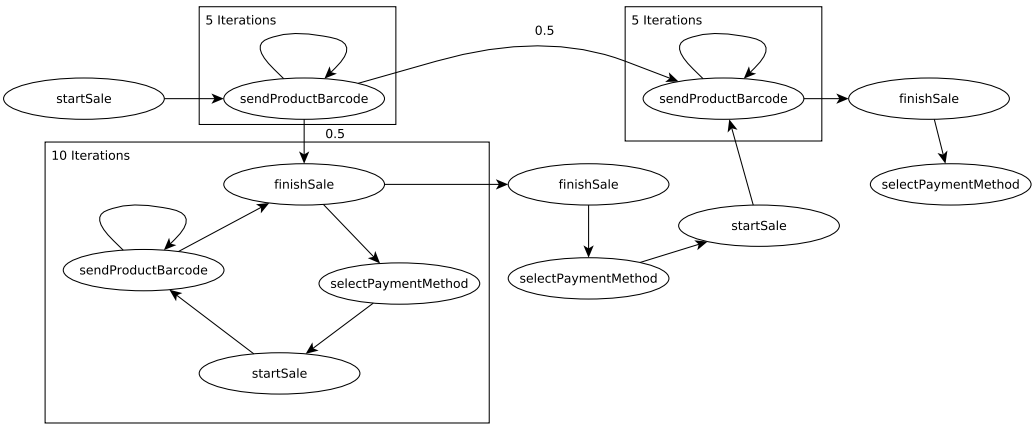


Figure 5.10. First clustering result of David Peter

Chapter 6

Related Work

In the domain of web usage mining different approaches are available to to aggregate user behavior based on the navigational pattern of the user. The approach Workload Extraction and Specification for Session-Based Application Systems (WESSBAS) [van Hoorn et al. 2014] extracts user behavior from monitored workload and created user behavior models. Theses models are used to generate workload for load testing. In contrary to our approach, WESSBAS uses relative models in the form of Markov chains. Markov chains facilitate the generation of user workload, but lacks the information of the calls of the transitions. Our approach uses absolute, which contain the number of transition calls. An absolute model can be transformed to an relative model straightforward. The WESSBAS behaviors contain the think time of the user for each transition, but do not provide additional information. Since our approach can contain arbitrary call-information, we are able to model think-time as well.

An implementation of the WESSBAS approach is presented in the masters thesis of David Peter [Peter 2016]. It implements the behavior models of the WESBASS approach for the iObserve framework. It uses relative behavior models represented by usage models of the Palladio Component Model (PCM) and consist of the navigational pattern of the user. The aggregation of user behavior is tightly bound to the used PCM models used in the iObserve framework. For the clustering the X-Means algorithm is used and not exchangeable via configuration. The input parameters for the clustering are chosen iteratively. After each clustering execution, the found number of clusters is taken as expected number of user group for the next clustering. In contrast, our approach is independently from the PCM models, which makes it individually configurable. Furthermore, we are not dependent on the X-Means clustering algorithm, which turned out to be unreliable (see Section 5.3.3).

An older approach to discover usage patterns is the approach of Gündüz and Özsu [Gündüz and Özsu 2003b]. Its underlying clustering compares user click-streams pairwise using a similarity metric. Highly similar user sessions are then aggregated and a click-stream tree is created. Each node in the tree represents a web-page and its children the next possible pages of a user to visit. The branches denote the probability of a user visiting the child node. The click-stream tree is used to predict user behavior. The tree based model is the main

difference to our graph-based approach. One advantage for the graph-based approach is its compactness compared to the click-stream tree, which may contain the same operation call multiple times and is more comprehensible when visualized.

For the monitoring of user behavior in social networks, Wang et al. [Wang et al. 2016] provide a solution to aggregate user behavior to a hierarchical behavior model. Their software aggregates the click-stream collected from HTTP logs with a self designed hierarchical clustering. The created clustering performs well in comparison to K-Means and a another hierarchical clustering methods. Furthermore, it provides a visualization that features different views on the created behavior models. The software of Wang et al. is designed as standalone software with low configurability and it is not connected to a deeper monitoring. Its analysis is based on the users HTTP click-streams extracted from the session logs. Our approach provides a higher configurability, i.e., exchangeable clustering or entry call filtering. Furthermore our approach allows to cluster events from deeper system layers, which can provide an abstracted reduced behavior.

Chapter 7

Conclusions

In this chapter we present the conclusion of our approach. First we summarize our evaluation results and our scientific work in Section 7.1. Then in Section 7.2 we list our technical work for this approach. Finally in Section 7.3 we present our ideas and suggestions for future improvements of the approach.

7.1 Summary of the Evaluation Results

The core goal of this thesis was the development of a user behavior clustering which includes information based on data provided by the user and the domain to achieve better clustering results. We evaluated this approach in two steps. In the first evaluation our aim was to find predesigned user type enriched with user specific data from generated workload in the JPetstore, whereas the second was a comparison with the approach of David Peter which does not use additional information.

The evaluation for the JPetstore has shown that it is possible to find predesigned user behavior in generated workload. The results of the clustering depend on the choice and the configuration of the clustering algorithm. In out evaluation scenario, we used the Weka implementation of the X-Means algorithm. We discovered that the algorithm does not fit to the clustering problem at hand, because it did not produce the expected results. This might be caused by the high dimensionality of our input vectors.

We implemented a concept for clustering user data. Clustering creates a new cluster every time it is executed. If the algorithm is easily affected by noise or small changes of the data set, it can occur that the results are highly different than expected. This can even be the case if the data changed only a bit. In our evaluation X-Means provided unsatisfactory results. This indicates that X-Means is not well suited for the automatic clustering of user behaviors.

7.2 Technical Contribution

This thesis provides an approach for the aggregation of user-behavior in the form of user-sessions for the iObserve framework. The approach is implemented as pipe-and-filter architecture supported by the TeeTime framework. Every processing step for the user-behavior generation is realized in its own filter, therefore, we have high modularity with

a low coupling. A summary over all filters is presented in Section 7.2.1. We designed our implementation for general purpose scenarios and thereby provided configuration classes, which are depicted in Section 7.2.2. To cluster behavior with call-information, we have to aggregate them from the monitoring component. Furthermore, we made a custom integration of our approach for the JPetstore and CoCoME example applications, which is explained in Section 7.2.4.

7.2.1 Created Filters for the Behavior Model Generation

To process the session stream provided by the analysis, we created a pipe-and-filter system with TeeTime. Through the use of composite stages we structured and encapsulated our service for an easier deployment. To integrate our implementation only the `TBehaviorModel` stage has to be instantiated and connected to a stage producing entry-call sequence models. The inner stages of the `TBehaviorModel` are exchangeable or reusable with low configuration effort, if necessary.

The substages of `TBehaviorModel` are the `TBehaviorModelPreprocessing`, which preprocesses the incoming sessions to instance vectors for the clustering, and the `TBehaviorModelAggregation`, which transforms the instance vectors into behavior models.
The `TBehaviorModelPreprocessing` is assembled by four stages. First the sessions are filtered in the `TEntryCallFilter`, then the `TBehaviorModelGeneration` and the `TBehaviorModelPreprocessing` transform the sessions into `BehaviorModelTable`-objects containing a graph representation of the user behavior. These tables are transformed into instance vectors for the clustering in the `TInstanceTransformation`. In the `TBehaviorModelAggregation` the vectors are clustered by the `TClusteringStage` and transformed to instances of the `BehaviorModel`. These models are then send to the visualization by the `TBehaviorModelVisualization`.

The `TClusteringStage` is usable with any self designed clustering algorithm, that accepts and produces Weka instances. We designed it to be reusable in other contexts of the iObserve framework.

7.2.2 Configuration Objects

We designed our approach for a general purpose, therefore we provide high level of configurability. A developer/operator can configure our service individually for his specific requirements. He can filter out non relevant operation calls with the `EntryCallFilterRules` and configure his own clustering algorithm to adapt to his input behavior by using the `IClustering`. The `BehaviorModelConfiguration` stores all these configurations.

7.2.3 Extension of the Kieker and iObserve Events for Clustering

To transport our call-information from the monitoring to the iObserve analysis, we extended the the `BeforeOperationEvent` and the `AfterOperationEvent` of Kieker as well as the

`EntryCallEvent` of iObserve to hold a string. The call-information are stored in these events as JSON string.

7.2.4 Instrumentation for the JPetstore and CoCoME

We instrumented the JPetstore with a modification of the `SessionAndTraceRegistrationFilter` of the iObserve servlet filters. The URLs of the JPetstore are defined by the actions they are calling internally. We deconstructed each URL string in the operation call with its parameters of the underlying system. The parameters are stored as call-information in an extended operation event with the signature of its operation call.

For the analysis of CoCoME and the JPetstore, we created filter rules to filter out non relevant operation calls. Additionally we created strategies for shortening the operation names in the visualization. Furthermore, we designed a strategy to find the representative call-information of an entry-call for the JPetstore.

7.2.5 Automatic Workload Generation Scripts

For the generation of user types for the JPetstore and CoCoME, we used the Selenium browser automation to execute the headless PhantomJS browser with scripts simulating user behavior. The script for each user type is recorded with the Selenium IDE and transformed into a Java class, where the behavior can be looped and reproduced.

7.2.6 Setup for the Comparison of Approaches

For the comparison of David Peters approach and our approach we created the composite stage `TBehaviorModelComparison`. This stage include instance of the `TEntryEventSequence` and `TBehaviorModel`, which are the implementations of the both approaches. To provide the same input for both approaches, we connect their input ports to an distributer, which sends both stages the same `EntryCallSequences`.

To visualize the output of the `TEntryEventSequence` stage, we created the `TUsageModelToBehaviorModel` stage and connect it to an instance of `TBehaviorModelVisualization`. Thereby, the created usage models of David Peters approach are transformed to behavior models and subsequently send to the visualization server.

7.3 Future Work

In this section we describe the possible future work of this approach. We divided the chapter based on the characteristic of the work. In Section 7.3.1 we describe all technical work, while were discussing possible scientific work in Section 7.3.2.

7.3.1 Technical Work

Start Node Detection When we are transforming the entry-call sequences to behavior model tables, we take each transition between two entry-calls and add it to the table. In this table, the order in which the entry-calls are added is not preserved. Therefore, we cannot tell which node of the behavior is the start node. However, knowing the start node is relevant to understand user behavior. In the future, a flag for the start of the behavior could be added to signalize the initial entry-call. This could also be realized with a call-information at the start node. If we add a flag or a call-information to determine the start of a behavior, we have to increase the dimension of the input vector for the clustering. Depending on the implementation of the clustering algorithm and the added number of dimension to the vector, this can have a negative effect on the results. Thus, we have a trade-off that has to be evaluated in the future.

Configure the Stages for a Live Environment The implementation of our approach is configured for a single execution of the analysis. The `TBehaviorModelGeneration` sends the created `BehaviorModelTable` to its output port, when the stage is terminating. Thereby, it would never send the table to the next stage in a live scenario. To use our system in a live environment, the `TBehaviorModelGeneration` needs a signal for sending the current behavior model table to an output port. This signal can be used to define a clustering interval. Every x `EntryCallSequenceModel`-instances a new prototype model table has to be created. With the new prototype, the x sequence models can be transformed to clustering instances and clustered by our implementation. Thereby, the user behavior models are renewed x `EntryCallSequenceModel`-instances and represent the current user behavior on the system.

Transfer Call-Information Encoding from the Monitoring to the Analysis The instrumentation of a system to get call-information for the operation events is the most complex part for a developer or operator of the system. In our implementation, we proposed to encode the call-information, since we can only cluster numerical vectors. We encode them during the monitoring and send the encoded call-information to the analysis. Not only is the instrumentation complicated but the overhead of the monitoring increases. Since we want to keep the overhead of the monitoring low for performance reasons, we should encode the call-information in the analysis. It would additionally give us the advantage that we have to encode in the analysis exclusively. When we decode the call-information after the clustering and send the decoded information to the visualization, the encoding is not handled. Thereby, an operator maintaining the system would see the same call-information go into the clustering and come out in the visualization, which could mitigate confusion.

Filter Call-Information In some cases we do not need all call-information of the extended entry-call events for the clustering. For example if we want to cluster users of the JPetstore

only by the call-graph and the category, a user is browsing without the call-information about viewed product and we cannot filter them out in the analysis. The `TEntryCallFilter` stage filters all entry-calls that are not relevant for the clustering, they are filtered out based on their operation signature. The stage can be extended to delete call-information from specific entry-call events.

7.3.2 Scientific Work

Context Information Based on System Knowledge In our implementation, user information is bound to entry-calls. Thereby, general information, i.e. age of a user, can only be added by abusing the behavior-model. The age of a user can be designed as an entry-call containing the age as call-information. The age would therefore be relevant for the clustering, but not displayed in the visualization, since only the edges of the behavior model are displayed. We either have to change the rules for the visualization or connect the age node via edge to the behavior-model graph. In the future, a concept for adding global call-information should be provided without the abuse of the entry-call node.

Alternative Clustering Algorithms The results of the clustering are highly dependent on the clustering algorithm and its configuration parameters (see Section 5.3). Further clustering algorithms should be evaluated and implemented to achieve improved clustering results than X-Means.

For a continuous adaptation to user-behavior, a concept for classification [DU 2010, chapters 6-7] of user groups could be useful. When using classification, we have predesigned groups and every new behavior is added to the group, which represents it the best. With each added behavior the user group changes which leading to ever-evolving user behavior models. In the future, the clustering stage could be exchanged by a classification stage dependent on the requirements of each specific use case to achieve the best results.

Bibliography

[Abbas 2008] O. A. Abbas et al. Comparisons between data clustering algorithms. *Int. Arab J. Inf. Technol.* 5.3 (2008), pages 320–325. (Cited on pages 24 and 41)

[Banck 2017] D. Banck. Live visualization and editing of user behavior iobserve. PhD thesis. Department of Software Engineering, University of Kiel, 2017. (Cited on pages 13 and 38)

[Banerjee and Ghosh 2001] A. Banerjee and J. Ghosh. Clickstream clustering using weighted longest common subsequences. In: *Proceedings of the web mining workshop at the 1st SIAM conference on data mining*. Volume 143. 2001, page 144. (Cited on page 21)

[Basili 1992] V. R. Basili. Software modeling and measurement: the goal/question/metric paradigm (1992). (Cited on page 19)

[Basili and Weiss 1983] V. R. Basili and D. M. Weiss. *A methodology for collecting valid software engineering data*. Technical report. DTIC Document, 1983. (Cited on page 18)

[Becker et al. 2007] S. Becker, H. Koziolek, and R. Reussner. Model-based performance prediction with the palladio component model. In: *Proceedings of the 6th international workshop on Software and performance*. ACM. 2007, pages 54–65. (Cited on page 11)

[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 pages 11, 12)

[Buschmann 1998] F. Buschmann. *Pattern-orientierte software-architektur: ein pattern-system*. Pearson Deutschland GmbH, 1998. (Cited on page 5)

[Dharmarajan and Dorairangaswamy 2016] K. Dharmarajan and D. M. Dorairangaswamy. Web usage mining: improve the user navigation pattern using fp-growth algorithm. *Elysium journal of engineering research and management (EJERM)* 3.4 (2016). (Cited on page 21)

[DU 2010] H. DU. *Data mining techniques and applications*. Cengage Learning EMEA, 2010. (Cited on pages 8, 25, and 71)

[Gündüz and Özsu 2003a] Ş. Gündüz and M. T. Özsu. A web page prediction model based on click-stream tree representation of user behavior. In: *Proceedings of the ninth ACM SIGKDD international conference on Knowledge discovery and data mining*. ACM. 2003, pages 535–540. (Cited on page 65)

[Gündüz and Özsu 2003b] Ş. Gündüz and M. T. Özsu. A web page prediction model based on click-stream tree representation of user behavior. In: *Proceedings of the ninth ACM SIGKDD international conference on Knowledge discovery and data mining*. ACM. 2003, pages 535–540. (Cited on page 65)

[Hall et al. 2009] M. Hall, E. Frank, G. Holmes, B. Pfahringer, P. Reutemann, and I. H. Witten. The weka data mining software: an update. *ACM SIGKDD explorations newsletter* 11.1 (2009), pages 10–18. (Cited on page 7)

[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 (2013). (Cited on pages 3 and 10)

[Heinrich et al. 2015] R. Heinrich, E. Schmieders, R. Jung, W. Hasselbring, A. Metzger, K. Pohl, and R. Reussner. Run-time architecture models for dynamic adaptation and evolution of cloud applications (2015). (Cited on page 10)

[Herold et al. 2008] S. Herold, H. Klus, Y. Welsch, C. Deiters, A. Rausch, R. Reussner, K. Krogmann, H. Koziolek, R. Mirandola, B. Hummel, et al. "Cocome-the common component modeling example". In: *The Common Component Modeling Example*. Springer, 2008, pages 16–53. (Cited on pages 15, 16)

[Jung 2013] R. Jung. *An instrumentation record language for kieker*. Technical report. Tech. rep. Kiel University, 2013. (Cited on page 11)

[Kephart and Chess 2003] J. O. Kephart and D. M. Chess. The vision of autonomic computing. *Computer* 36.1 (2003), pages 41–50. (Cited on page 10)

[Node.js] *Node.js*. Node.js Foundation. URL: https://nodejs.org. (Cited on page 17)

[Pelleg and Moore 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 8 and 24)

[Peter 2016] D. Peter. Observing and modeling workload characteristics of dynamic cloud applications. PhD thesis. Department of Informatics Institute for Program Structures and Data Organization (IPD), 2016. (Cited on pages 4, 11, 15, 21, 36, 43, 48, 50, 61, and 65)

[PhantomJS] *Phantomjs*. Ariya Hidayat. URL: http://phantomjs.org/. (Cited on page 17)

[Selenium] *Selenium browser automation*. Selenium. URL: http://www.seleniumhq.org/. (Cited on pages 17, 50, and 54)

[Southekal 2017] P. H. Southekal. *Data for business performance: the goal-question-metric (gqm) model to transform business data into an enterprise asset*. Technics Publications, 2017. (Cited on pages 19, 20)

[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 23)

[Suthar and Oza 2015] P. Suthar and B. Oza. A survey of web usage mining techniques. *International Journal of Computer Science and Information Technologies* 6.6 (2015), pages 5073–5076. (Cited on pages 24, 25, and 41)

[*TeeTime Framework Webpage*] *TeeTime framework webpage*. `http://teetime-framework.github.io/`. Accessed: 2016-11-17. (Cited on pages 3 and 5)

[Urma 2014] R.-G. Urma. Processing data with java se 8 streams. *Java Magazine March/April 2014* (2014), pages 51–56. (Cited on page 5)

[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 ACM/SPEC International Conference on Performance Engineering*. ACM. 2012, pages 247–248. (Cited on page 11)

[Van Solingen et al. 2002] R. Van Solingen, V. Basili, G. Caldiera, and H. D. Rombach. Goal question metric (gqm) approach. *Encyclopedia of software engineering* (2002). (Cited on page 19)

[Van Hoorn et al. 2008] A. van Hoorn, M. Rohr, and W. Hasselbring. Generating probabilistic and intensity-varying workload for web-based software systems. In: *SPEC International Performance Evaluation Workshop*. Springer. 2008, pages 124–143. (Cited on page 14)

[Van Hoorn et al. 2014] A. van Hoorn, C. Vögele, E. Schulz, W. Hasselbring, and H. Krcmar. Automatic extraction of probabilistic workload specifications for load testing session-based application systems. In: *Proceedings of the 8th International Conference on Performance Evaluation Methodologies and Tools*. ICST (Institute for Computer Sciences, Social-Informatics and Telecommunications Engineering). 2014, pages 139–146. (Cited on pages 14, 21, and 65)

[Verma et al. 2012] M. Verma, M. Srivastava, N. Chack, A. K. Diswar, and N. Gupta. A comparative study of various clustering algorithms in data mining. *International Journal of Engineering Research and Applications (IJERA)* 2.3 (2012), pages 1379–1384. (Cited on page 24)

[Vögele et al. 2015] C. Vögele, A. van Hoorn, and H. Krcmar. Automatic extraction of session-based workload specifications for architecture-level performance models. In: *Proceedings of the 4th International Workshop on Large-Scale Testing*. ACM. 2015, pages 5–8. (Cited on pages 14, 15)

[Wang et al. 2016] G. Wang, X. Zhang, S. Tang, H. Zheng, and B. Y. Zhao. Unsupervised clickstream clustering for user behavior analysis. In: *Proceedings of the 2016 CHI Conference on Human Factors in Computing Systems*. ACM. 2016, pages 225–236. (Cited on page 66)

[Wulf et al. 2014] C. Wulf, N. C. Ehmke, and W. Hasselbring. Toward a generic and concurrency-aware pipes & filters framework (2014). (Cited on pages 5 and 27)

Appendix

Code Repository

`https://github.com/research-iobserve/iobserve-analysis`
Branch : cdor-userbehavior
Commit : d444f00037ce1cc214a8d8a28a2b101c4c1a4510