Extended Trace-based Task Tree Generation

Patrick Harms, Steffen Herbold, and Jens Grabowski
Institute of Computer Science
University of Göttingen
Göttingen, Germany
E-mail: {harms,herbold,grabowski}@cs.uni-goettingen.de

Abstract—**Task trees are a well-known way for the manual modeling of user interactions. They provide an ideal basis for software analysis including usability evaluations if they are generated based on usage traces. In this paper, we present an approach for the automated generation of task trees based on traces of user interactions. For this, we utilize usage monitors to record all events caused by users. These events are written into log files from which we generate task trees. The generation mechanism covers the detection of iterations, of common usage sequences, and the merging of similar variants of semantically equal sequence. We validate our method in two case studies and show that it is able to generate task trees representing actual user behavior.**

Keywords-task tree generation, usage-based, traces, task tree merging.

I. Introduction

Task trees are an established method to model user interactions with websites. They can be created manually at design time or automatically, e.g., based on recorded user actions [1] or based on existing Hyper-Text Markup Language (HTML) source code [2]. When created manually at design time [3], the structure of task trees reflects the user interactions as intended by the interaction designer [4]. Task trees can also be used for comparing expected and effective user behavior as a basis for a semi-automatic usability evaluation [5]. When they are not generated based on recorded user actions, task trees do not describe effective user behavior but either expected or possible user behavior.

In this paper, we present an approach for automatically generating task trees based on recordings of user interactions. This approach does not require a manual marking of task executions in the recorded data before the task generation making it easily applicable to larger data sets and differentiating it from other approaches. The generated task trees represent the effective behavior of users. They can, therefore, be used for a detailed analysis of the usage of a software what is the major goal we intend to achieve based on our approach. For example, in other work we utilize the generated task trees for an automatic usability evaluation of a website [6]. The results of a usage analysis are used for optimizing software with respect to the user's needs. Throughout the remainder of this paper, we use the analysis of websites as a running example. However, our approach is designed for event-driven software in general including desktop applications.

The approach in this paper is an extension of our work described in [1]. The extension covers mainly the detection and merging of similar generated sequences. We provide details about the challenges introduced by the merging process and extended the case studies section to also evaluate the merging results. Furthermore, we added sections describing in more details why and how the recorded user actions need to be post-processed and how we detect and handle common elements on different pages of a website, e.g., menu structures, to improve the detection of tasks. Finally, we extended the related work section to compare our approach and the resulting task trees in more details with other approaches.

The remainder of this paper is structured as follows. First, we introduce our approach and the respective terminology in Section II. Then, we describe an implementation in Section III. In Section IV, we present two case studies in which we tested the feasibility of our approach and discuss the case study results in Section V. Finally, we refer to related work in Section VI and conclude with an outlook on future work in Section VII.

II. Trace-based Task Tree Generation

The goal of our approach is to generate task trees based on recorded user actions. In this section, we introduce the details of the approach that consists of four major steps: user interaction tracing, data preparation, detection of sequences and iterations, and merging of similar sequences. These major steps are shown in Figure 1. We commence with the definition of terms that we use in this paper. In the subsequent sections, we describe the details for the four major steps.

A. Terminology

Users utilize a website by performing elementary *action*s. An action is, e.g., clicking with the mouse on a button, typing some text into a text field, or scrolling a page. Actions cause *event*s to occur on a website, also known as Document Object Model (DOM) events. An event is characterized by a *type* that denotes the kind of event and, hence, the type of action that causes the event. Furthermore, an event refers to a *target* indicating the element of a website on which the corresponding action was executed and where the event was observed. For example, clicking with a mouse on a link (action) causes an event of type `onclick` with the link as its target. Typing a text into a text field causes an event of type `onchange`

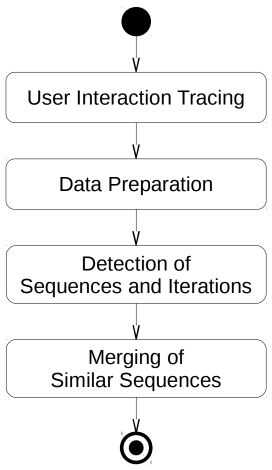

User Interaction Tracing
Data Preparation
Detection of Sequences and Iterations
Merging of Similar Sequences

Figure 1. Overall process for generating task trees

with the text field as its target. Events are representations of actions. For each action there is a mapping to an event caused by performing the action.

To execute a specific *task* on a website, users have to perform several actions. For example, for logging in on a website, users must type in a user name and a password into two separate text fields and click on a confirmation button.

Tasks and actions can be combined to form higher level tasks. For example, the task of submitting an entry on a forum website comprises a *subtask* for logging in on the website as well as several actions for writing the forum entry and submitting it. Therefore, tasks and actions form a tree structure called a *task tree*. The leaf nodes of a task tree are the actions users must perform to fulfill the overall task. The overall task itself is the root node of the task tree. The intermediate nodes in the task tree define a structure of subtasks for the overall task.

A task defines a *temporal relationship* for its children, which specifies the order in which the children (subtasks and actions) must be executed to fulfill the task. Different task modeling approaches use different temporal relationships [7]. In our work, we consider the temporal relationships *sequence*, *iteration*, *selection*, and *optional*. If a task is a sequence, its children are executed in a specified order. If a task is an iteration, it has only one direct child, which can be executed one or more times. If a task is a selection, only one of its children can be performed. If a task is an optional, it has only one child whose execution can be skipped. A leaf node in a task tree has no children and does, therefore, not define a temporal relationship.

An example for a task tree is shown in Figure 2. It represents the actions to be taken to perform a login on a website. The actions are the leaf nodes. The temporal relationships of their parent nodes define the order in which the actions have to be performed. The task starts with an iteration of a selection. The possible variants are entering a user name or a password in the respective fields. Users may enter and change their user name

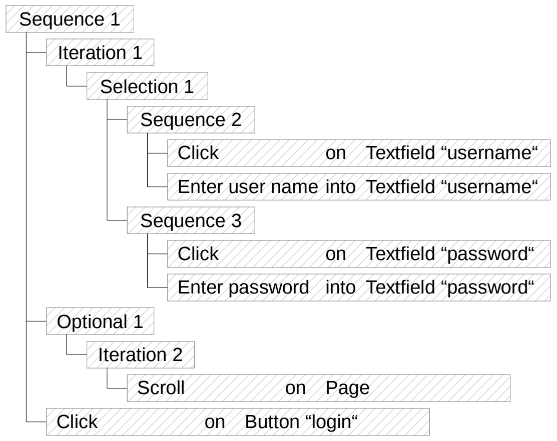

Sequence 1
Iteration 1
Selection 1
Sequence 2
Click on Textfield “username“
Enter user name into Textfield “username“
Sequence 3
Click on Textfield “password“
Enter password into Textfield “password“
Optional 1
Iteration 2
Scroll on Page
Click on Button “login“

Figure 2. Example for a task tree

and password several times. Optionally, users scroll before completing the overall task by clicking the login button.

The execution of a task is called a *task instance*. A task instance is a tree structure similar to that of the corresponding task. It reflects in detail how a task and its subtasks were executed. Each node in a task instance refers to the task of which it is an instance. The concrete structure of a task instance depends on the temporal relationships of the corresponding task. For example, an iteration has only one child, but an instance of an iteration (i.e., an *iteration instance*) has as many children as the single child of the iteration was executed. In contrast, a selection has several children but a *selection instance* has only one child that is an instance of the executed child, i.e., the chosen variant, of the selection.

An example of a task instance for the task tree in Figure 2 is shown in Figure 3. For executing the task *Sequence 1*, first the *Iteration 1* is executed two times indicated by its two children. Both children are instances of the child of *Iteration 1*, i.e., *Selection 1*. In the first instance of *Selection 1*, the user selected *Sequence 2* to be executed, in the second instance, *Sequence 3* was performed. To finalize the instance of *Sequence 1*, the user did not scroll before clicking the login button, which is indicated by an instance of *Optional 1* having no child instance.

A simplified representation of a task instance is a *flattened task instance*. A flattened task instance is an ordered list of the actions that were executed. This is identical to listing the leaf nodes of a task instance.

B. User Interaction Tracing

The first step in our approach is tracing user actions on a website. This is done by recording the events caused by the actions of users. We achieve this by integrating a monitoring module into the website. This module is invisible to the users and has minimal effect on the implementation, performance, and stability of the website [8]. The module can also be configured to ensure privacy by dropping user data from the recorded events. The resulting sequence of events is encrypted, sent to a server, and stored in a log file. A recorded sequence of events is called a trace.

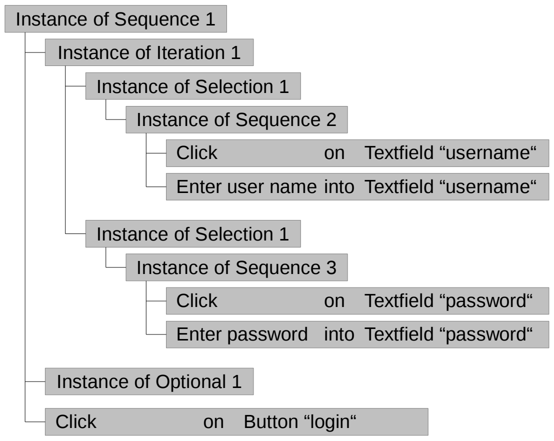


Figure 3. Example for an instance of the task tree in Figure 2

1.	Left mouse button click	on	Textfield with id username
2.	Text input „usr“	on	Textfield with id username
3.	Left mouse button click	on	Textfield with id username
4.	Text input „user“	on	Textfield with id username
5.	Left mouse button click	on	Textfield with id password
6.	Text input „“	on	Textfield with id password
7.	Left mouse button click	on	Button with name „login“

Figure 4. Example for a trace

A simplified example of a trace is shown in Figure 4. It lists the events recorded for a login of a single user on a website. The login comprises the entering of the user name and the password in the respective text fields, as well as a confirmation by clicking on the login button. As the user initially entered a wrong user name, it is reentered a second time. The user does not scroll before confirming the login.

C. Data Preparation

The second step in our approach is the preparation of the data gathered from tracing the user interactions. This includes correcting the recorded events and checking the structure of the website for common page elements to identify identical user actions on different pages of a website. We describe this data preparation in the following subsections.

1) Trace Post-Processing: Traces can have structural abnormalities that are unnatural for a user action. These abnormalities can be caused by the high level of detail on which the events are recorded, by the event types, or by the technology used for event recording. A typical example is that a technology provides events indicating a keyboard focus change separately to key strokes. Hence, the target of a key stroke event is influenced by the last preceding keyboard focus change event. In addition, traces may contain several events that together represent a single user action. For example, two subsequent clicks on the same website element in a short time period represent a double click.

Due to these abnormalities, we perform a post-processing of the traces before the generation of task trees. The post-processing can be automated, as the structural abnormalities in the traces always follow a specific pattern. For example, for each key stroke event, we check the last preceding keyboard focus change event and adapt the target accordingly. Afterwards, we drop all focus change events from the traces. Overall, we perform the following post-processing:

- *Detection of double clicks:* Two subsequent click events with the left mouse button on the same website element within a time frame of 500 milliseconds are transformed into a double click event.
- *Correction of the target of key stroke events:* The target of key stroke events is set to the target of the last preceding keyboard focus change event.
- *Correction of tab key navigation:* When navigating from one text field to another, two events are generated: a key stroke event for hitting the tab key and a value change for the first text field. These events are always recorded in reverse order (first tab key stroke, then value change) although logically the value change happened before the tab key stroke. In these cases, the order of both events is switched.

2) Common Page Elements: Nowadays, websites are composed of several pages all having a similar layout. For example, all pages of a website contain the same navigation menu made up of the same links. This means, that specific elements (links, images, buttons, etc.) reappear on all pages of a website. Although invisible for the user, these elements are in fact distinct instances of the same element. We call these *common page elements* as they are common to several pages of a website. On contrary, there are elements on different pages that reside at the same or similar location but are semantically distinct. Examples are form elements of different forms on different pages positioned similarly for design consistency.

When tracing user actions, we respect common page elements. We consider actions on common page elements as identical if they were performed on the same common page element. Otherwise, we consider them as distinct. This is important for the subsequent task tree generation. Without this consideration, generated tasks would be considered distinct although semantically they are not.

Elements on different pages are considered common, if they have the same id. The ids are defined by assigning them to nodes in the HTML Document Object Model (DOM). Usually, this is automatically ensured when using modern content management systems.

If a website does not make use of identical ids for common page elements, we add and harmonize the ids subsequently and automatically. For this we first create a mapping between HTML DOM nodes and the ids they should have. To identify an HTML DOM node, we use the page it belongs to and the complete path of nodes through the DOM of the page pointing to the node. A path through the DOM must be unambiguous. Hence, on every part of such a path, we consider the HTML tag of the node, its id if there is one, and its index with respect to the children of the parent node. An example for such a path is the following:

page1/html[0]/body[0]/table(id=tab_1)/tr[2]/td[3]/img[0]

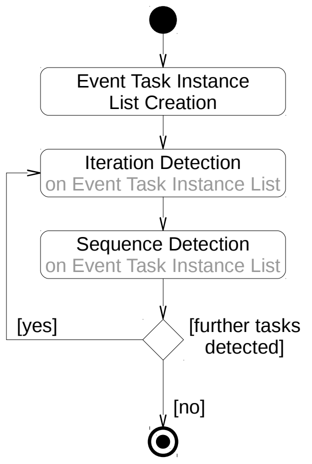

Event Task Instance List Creation
Iteration Detection on Event Task Instance List
Sequence Detection on Event Task Instance List
[yes]
[further tasks detected]
[no]

Figure 5. Basic process for detecting sequences and iteration

This path identifies an image embedded in a table on page “page1”. The image is located in the table with id “tab_1” in the third row (“tr[2]”) and the fourth column (“td[3]”). The image is the first child of the respective table cell.

For each common page element, we create a mapping between the path to the corresponding HTML DOM node and the id it should carry. Afterwards, we update the events in a trace. For each event, we check if the referenced target matches an entry of the mappings. If there is an entry, we update the information about the referenced target to include the id defined in the mapping. To be able to perform a lookup in the mapping, the target references in the events contain all required information including the page the target resides on and its position in the DOM. Eventually, all events having targets with the same id are considered to be recorded on the same common page element.

D. Detection of Sequences and Iterations

The third step of our approach is the detection of sequences and iterations. This step consists of three substeps: event task instance list creation, iteration detection, and sequence detection. The substeps may be executed several times to fully detect all sequences and iterations. The basic execution of these substeps is shown in Figure 5. In the following subsections, we describe these individual substeps and their repeated execution. For this, we first introduce the levels of design, which are important for structuring task trees. We then describe the creation of the initial task trees consisting only of sequences and iterations. Finally, we introduce some characteristics of these initial task trees.

1) Levels of Design: When designing Graphical User Interface (GUI)s, four levels of design are considered: conceptual design, semantic design, syntactical design, and lexical design. They are shown in Figure 6. The conceptual design describes the types of entities that are editable with a software [9], as well as their relationships [10]. For example, in a system for managing addresses, addresses and persons are the entity types. These entity types are related, because a person may be assigned zero or more addresses.

Conceptual Design	Types of entities and their relationships
Semantic Design	Functions to modify entities
Syntactical Design	Steps to take for executing functions on entities
Lexical Design	Physical execution of steps to execute functions on entities

Figure 6. Levels of design

The semantic design specifies functions to edit the entities defined in the conceptual design [9]. For the address management example, this includes adding, editing, and deleting addresses and persons. The syntactical design specifies the steps to execute a function defined in the semantic design [9]. For example, adding a new address is comprised of steps like adding a street name, a city, and a zip code. At the most detailed level, the lexical design specifies means of physically performing steps defined in the syntactical design [9]. In the example, defining a street of an address includes clicking on the respective text field and typing the street name.

In our approach, we map the semantic, syntactical, and lexical levels of design onto task trees. For each function specified in the semantic design, there exists at least one task for executing that function. Hence, there is at least one task tree for each function in the semantic design. The syntactic design is a decomposition of functions into individual steps for function execution. This decomposition corresponds to the definition of subtasks and their temporal relationships within task trees. The actions on the lexical level of design are represented through the leaf nodes of task trees. As we record the events mapped to the respective actions, we refer to the leaf nodes as *event tasks*. Event tasks are considered tasks with the constraint of not having children and not defining a temporal relationship.

2) Event Task Instance List Creation: Using the basic mapping of task trees to the levels of design, we create task trees starting from the leaf nodes, i.e., from the event tasks. For each event in a trace, we generate an instance of an event task. All event task instances are stored in an ordered list. The order in the list is given by the order in which the respective events were recorded. An example is shown in Figure 7a where each grey rectangle denotes an event task instance and the arrows denote their order. The letters of the event task instances denote the respective event task. If this letter is the same for several event task instances, it indicates that the same event task, i.e., the same action, was executed at distinct times.

3) Iteration Detection: The ordered list of event task instances may contain subsequent instances of the same event task. For example, the user might have clicked several times on the same button. Such tasks are represented in task trees as iterations. Therefore, we scan the list of event task instances for iterations of identical tasks. If we observe an iteration, we generate a new task node of type iteration. The single child of this task node becomes the iterated event task.

Afterwards, we scan the ordered list of event task instances for instances of the iterated event task. We replace all instances

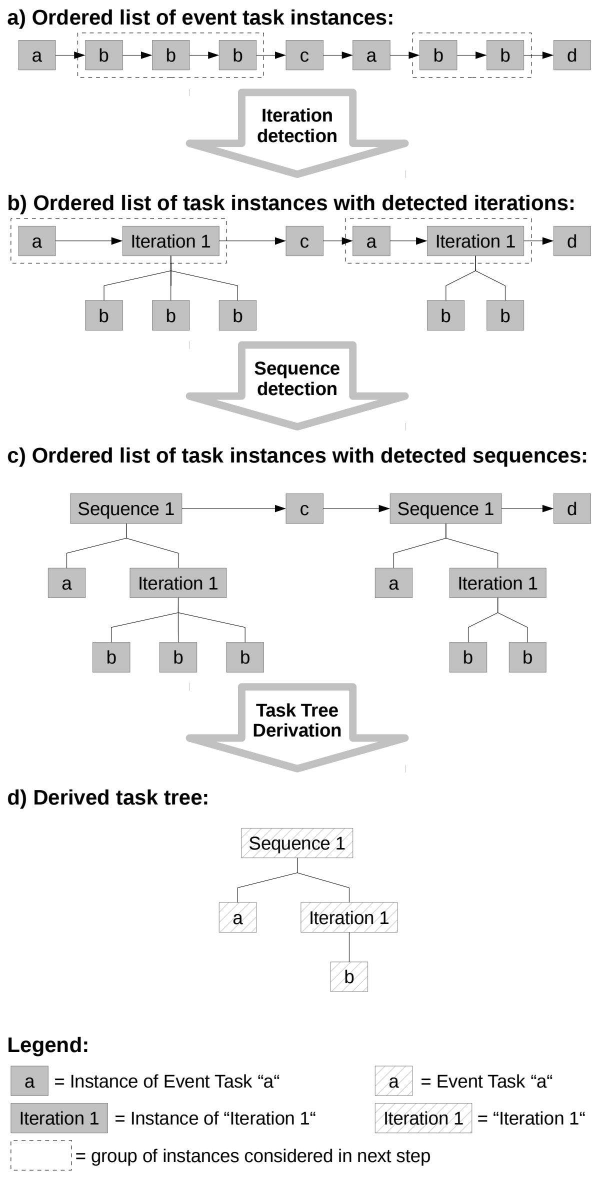

a) Ordered list of event task instances:
a
b
b
b
c
a
b
b
d
Iteration detection
b) Ordered list of task instances with detected iterations:
Iteration 1
Sequence detection
c) Ordered list of task instances with detected sequences:
Sequence 1
Task Tree Derivation
d) Derived task tree:
Legend:
= Instance of Event Task "a"
= Event Task "a"
= Instance of "Iteration 1"
= "Iteration 1"
= group of instances considered in next step

Figure 7. Example for the detection of iterations and sequences

of the iterated event task with instances of the new iteration. If the event task occurs only once, it is replaced by an iteration instance getting the replaced event task instance as its single child. If the event task occurs more than once subsequently, the subsequent instances are also replaced by a single iteration instance. This instance gets all replaced event task instances as its children. An example for such a replacement is shown in Figure 7a and 7b where Figure 7a depicts the event task instance list before the replacement and Figure 7b shows the event task instance list after the replacement. There, Event Task *b* is iterated several times (denoted by the dotted boxes in Figure 7a. We replace these instances in the ordered list with iteration instances having the replaced event task instances as their children.

4) Sequence Detection: After the iteration detection, we scan the list of task instances for multiple occurrences of the same subsequences. For the subsequence occurring most often and which is, therefore, most likely an occurrence of a logical subtask, we generate a new task node of type sequence. Its children are the tasks belonging to the identified subsequence. Each occurrence of the identified subsequence in the task instance list is replaced with an instance of the new sequence. Each instance gets as children the task instances to be replaced by the sequence instance. An example is shown in Figure 7 where Figure 7b depicts the task instance list before the replacement and Figure 7c shows the task instance list after the replacement. There, the subsequence of Event Task *a* and Iteration *1* occurs most often (two times) and is, therefore, replaced through instances of a sequence representing this subsequence.

The subsequences replaced through the sequence detection can have any length. At the minimum, they have a length of two. Our algorithm searches for the longest subsequences occurring most often and replaces it accordingly. If several subsequences have the same maximum occurrence count, we replace only the longest one. If several subsequences have the same maximum count and the same maximum length, we replace only the subsequence occurring first in the ordered list.

5) Alternating Repetition of Detections: The iteration and sequence detection on the ordered list of task instances are repeated alternately until no more replacements are possible. This is also visualized in Figure 5. Each time an iteration detection is done, all iterations are detected and replaced. This also includes iterations of detected sequences. For each sequence detection the longest sequence occurring most often is replaced. A detected sequence may include already detected sequences and iterations. For example, in Figure 7c the detected sequence contains a previously detected iteration.

In each alternating repetition of the iteration and sequence detection, the ordered list of task instances becomes shorter. This is because several task instances in the list are replaced by single sequence or iteration instances. But the replaced task instances as well as their order are preserved by making them children of their respective replacement. Hence, no details of the recorded events are lost.

If no more iterations or sequences are detected, the algorithm stops as shown in Figure 5. The resulting task instance list contains instances of detected task trees as well as event task instances that were neither iterated nor part of a sequence occurring more than once. Based on the instances of the detected task trees, we can derive the raw task tree. For the example in Figure 7a-c, the detected task tree is shown in Figure 7d. The detected task trees represent the lexical, syntactical and semantic level of design. The more recorded events are processed, the more complex and deeper task trees are created.

Within a recording of only one user session, specific subsequences occur only once. An example is the login process, which is usually done only at the beginning of a recorded user session. With our approach, such regularly occurring subsequences would not be detected if only one session was considered. Therefore, we consider several sessions of different users at once for counting the number of occurrences of subsequences. Due to this, we also detect subsequences occurring seldom in individual sessions but often with respect to all recorded users of the website.

6) Characteristics of Detected Tasks: The task trees created so far consist only of sequences and iterations. Each detected task is characterized by several aspects with respect to the recorded events. First, each task t is associated a set of recorded events $r(t)$ based on which its structure was generated. For example, in Figure 7, the task *Iteration 1* is generated based on all instances of Event Task b and, hence, based on the corresponding events. Second, there is a function $depth(t)$ that returns the depth of a task t. The depth is the number of levels a task has where the task itself is the first level, its children are the second, its grandchildren are the third, etc. The last level contains only event tasks and hence no further children exist. For example, the depth of task *Sequence 1* in Figure 7c is 3. Finally, it is possible to determine all instances $i(t)$ of a task t as they were generated during the task detection process.

E. Merging of Similar Sequences

The tasks detected through the alternating iteration and sequence detection follow strict structures and contain only iterations and sequences. Due to this strictness, two distinct tasks can be similar to each other and may describe the same overall task being executed in two slightly different variants. A typical example is the login process where some users use the tab key and some users use a mouse click to navigate from the user name field to the password field. A simplified example of two similar tasks is shown in Figure 9a. Both tasks start with Event Task a, have intermediate executions of Event Tasks c and e, and end with Event Task h.

To reduce the number of similar tasks, we perform the forth and final major step of our approach for generating task trees based on recorded user actions that is the merging of similar sequences (see Figure 1). This step also consists of several substeps: detection of similar sequences, determination of sequences to merge, adaptation of flattened task instances, iteration detection, and sequence detection. These substeps may be repeated several times depending, e.g., on how many similar sequences are detected. The order of execution of these substeps is shown in Figure 8. For each of these substeps and their repetition, we provide respective details in the following sections.

1) Detection of Similar Sequences: The first substep to merge similar sequences is to detect them (see substep one in Figure 8). For this, we compare all sequences with each other and calculate a similarity metric for each pair. For this, we first generate for each sequence t an ordered list $l(t)$ of event tasks covered by the sequence. This list contains the event tasks in the order they would be executed, if the sequence was executed with a minimum of event tasks, i.e., with all iterations being executed only once. This list is similar to the smallest flattened instance of the sequence with the distinction that it contains event tasks and not their respective instances. We create $l(t)$ by performing a depth first traversal of the structure of t and storing all leave nodes in a list in the order they were visited. The two lists of event tasks belonging to the sequences in Figure 9a are shown in Figure 9b.

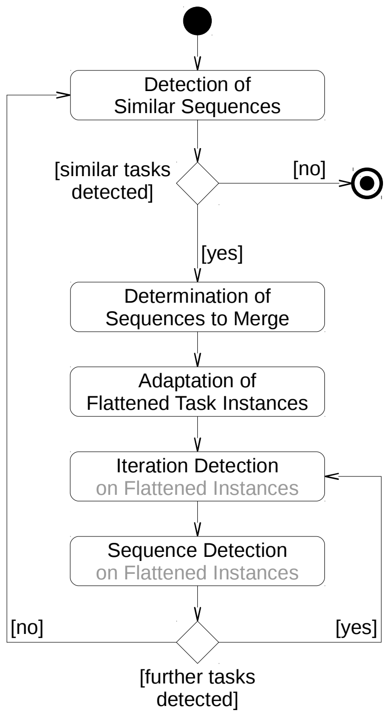

Detection of Similar Sequences
[similar tasks detected]
[no]
[yes]
Determination of Sequences to Merge
Adaptation of Flattened Task Instances
Iteration Detection on Flattened Instances
Sequence Detection on Flattened Instances
[no]
[yes]
[further tasks detected]

Figure 8. Basic process for merging similar sequences

In the next step, we compare these lists of event tasks with each other using Myers diff algorithm [11], which we adapted to compare lists of event tasks instead of strings. This diff can be seen as a function $D(l(t_1), l(t_2))$ that calculates a list of n deltas $d_1 \dots d_n$ between the two lists of event tasks determined for t_1 and t_2. The types of deltas that are detected are:

- *insert*: one or more subsequent event tasks occur only in the second list at a specified position;
- *delete*: one or more subsequent event tasks occur only in the first list at a specific position; and
- *change*: one or more subsequent event tasks in the first list are replaced by one or more subsequent event tasks in the other list.

For each delta d, the number of event tasks making up the delta is defined as $e(d)$. The three deltas determined for the sequences in Figure 9a are shown in Figure 9b. Based on the deltas, we calculate the similarity metric $s(t_1, t_2)$ of two sequences t_1 and t_2. This metric is calculated as the number of event tasks belonging to the determined deltas divided by the number of all event tasks of both sequences:

$$s(t_1, t_2) = \frac{\sum_{i=1}^{n} e(d_i)}{|l(t_1)| + |l(t_2)|} \tag{1}$$

In this calculation, we have to do a special consideration for scrolls. If an event task is a scroll, it must always be considered

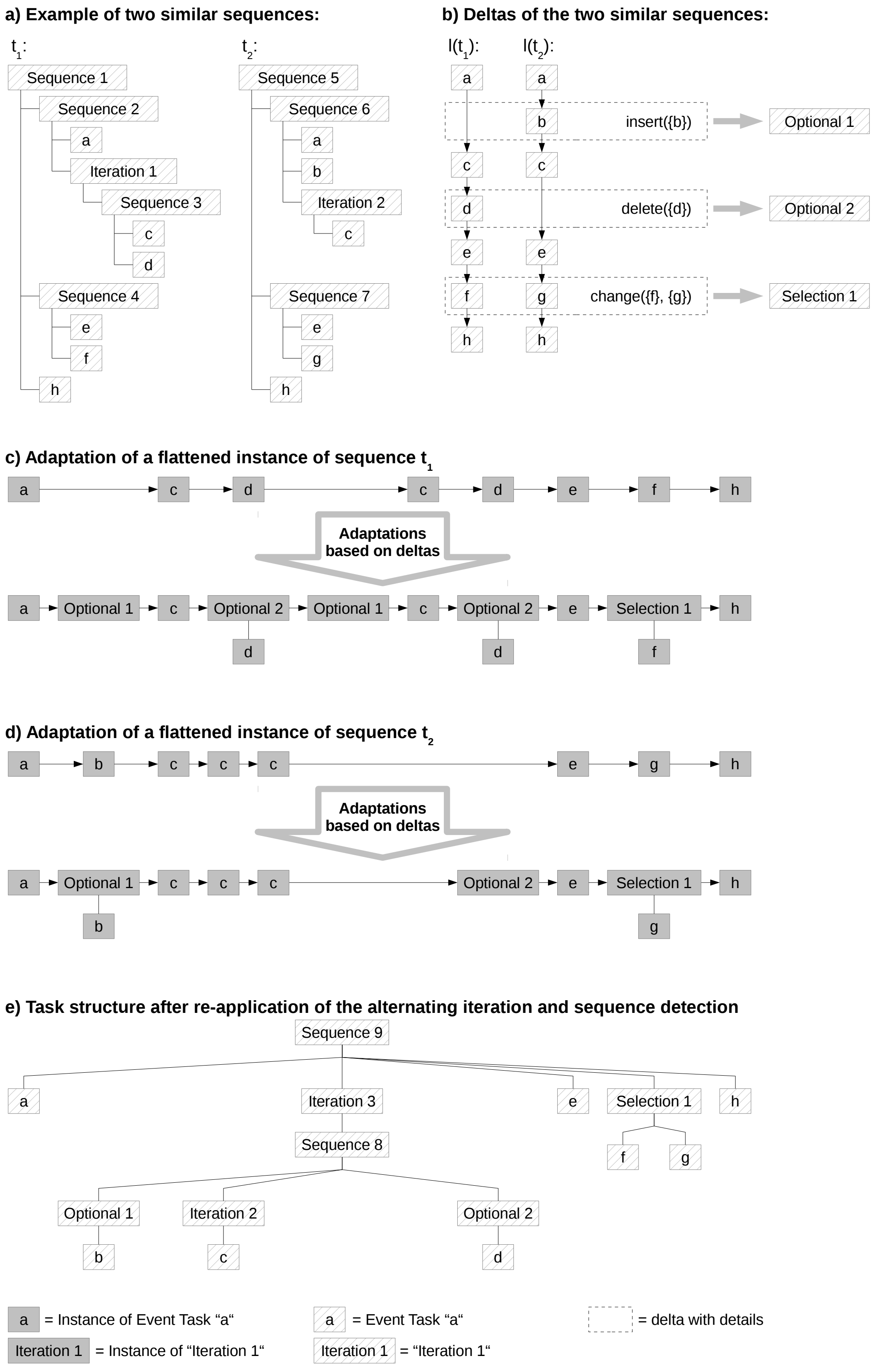

a) Example of two similar sequences:
t1:
Sequence 1
Sequence 2
a
Iteration 1
Sequence 3
c
d
Sequence 4
e
f
h
t2:
Sequence 5
Sequence 6
a
b
Iteration 2
c
Sequence 7
e
g
h
b) Deltas of the two similar sequences:
l(t1):
l(t2):
insert({b})
Optional 1
delete({d})
Optional 2
change({f}, {g})
Selection 1
c) Adaptation of a flattened instance of sequence t1
Adaptations based on deltas
d) Adaptation of a flattened instance of sequence t2
Adaptations based on deltas
e) Task structure after re-application of the alternating iteration and sequence detection
Sequence 9
Iteration 3
Sequence 8
Selection 1
Optional 1
Iteration 2
Optional 2
= Instance of Event Task "a"
= Event Task "a"
= delta with details
= Instance of "Iteration 1"
= "Iteration 1"

Figure 9. Example of two distinct but similar tasks generated by the alternating iteration and sequence detection and the process for merging them

distinct during the calculation of $s(t_1, t_2)$ independent of belonging to a delta or not. The reason for this is that scrolls do not have a semantic meaning in fulfilling a task. But two similar tasks should only be considered similar if their semantic meaning is similar. This should not be influenced by a vast occurrence of semantically unimportant event tasks like scrolling. Hence, we adapt the calculation of $s(t_1, t_2)$ accordingly. Let $o(d)$ be the number of scroll event tasks belonging to delta d and let $o(t_1, t_2))$ be the number of scroll event tasks belonging to $l(t_1)$ and $l(t_2)$. Through the following calculation of $s(t_1, t_2)$, we ensure that scrolls are not considered:

$$s(t_1, t_2) = \frac{o(t_1, t_2) + \sum_{i=1}^{n} (e(d_i) - o(d_i))}{|l(t_1)| + |l(t_2)|} \qquad (2)$$

Through this calculation, $s(t_1, t_2)$ is smaller the more similar the sequences t_1 and t_2 are. If $l(t_1)$ is identical to $l(t_2)$ and if both lists do not contain any scrolls, $s(t_1, t_2)$ evaluates to 0 indicating the highest possible similarity between two sequences. However, as $s(t_1, t_2)$ does not consider subtask structures, the sequences may still be different with respect to execution order.

2) Determination of Sequences to Merge: After we calculated $s(t_1, t_2)$ for each pair of sequences p, we determine those pairs that have to be merged. This is the second substep for merging similar sequences as shown in Figure 8. Merging is only useful for sequences having a minimum level of similarity, i.e., $s(t_1, t_2)$ should be below or equal to a specific border that we call s_{max}. For merging, we determine only those pairs

1) for which $s(t_1, t_2) \leq s_{max}$
2) whose deltas are neither at the beginning nor at the end of $l(t_1)$ and $l(t_2)$ to ensure that deltas are always considered in their entirety
3) for which $s(t_i, t_j)$ is minimal and that are, hence, the pairs of the most similar sequences
4) for which none of the sequences is a direct or indirect parent of any sequence of another pair

For the remaining set P of pairs, we determine the set of sequences $T = t_1 \ldots t_2$ that are referred to by more than one pair in P. If there are no such sequences ($T = \emptyset$), we merge all pairs in P. If there are such sequences, we perform a further filtering based on them and consider only pairs of which at least one task is in T further. This is required to ensure that in such cases the sequences in T are merged always in the same order. For each sequence $t \in T$, we merge only the pair referring to t where the following characteristics of the pair are maximized or minimized in the given order in comparison to all other pairs referring to t:

- $r(t_1) + r(t_2)$ is maximized, i.e., both sequences of the pair were generated on a maximum of recorded events;
- $|i(t_1)| + |i(t_2)|$ is maximized, i.e., both sequences of the pair were most often executed;
- $depth(t_1) + depth(t_2)$ is minimized, i.e., both sequences of the pair have the smallest depth;
- the sum of the instances of both sequences as well as all their subtasks is maximized (subtasks are also executed in other task contexts and, hence, their number of instances is typically higher than the number of instances of a parent task);

If there are several pairs referring to a $t \in T$ for which these characteristics are maximized or minimized, we throw an exception and break up.

3) Adaptation of Flattened Task Instances: The result of the determination of similar sequences is a list of independent pairs of similar sequences. In the third substep of merging similar sequences (see Figure 8), we perform a merge for each of these pairs. The merging of a pair is done based on the flattened instances of both sequences t_1 and t_2 and on the deltas $D(l(t_1), l(t_2))$ determined for the pair. For this, the flattened instances of both sequences are created. Then, they are adapted based on the deltas to include instances of optionals and selections to reflect the deltas. Basically, we

- create instances of optionals to reflect insert and delete deltas as well as
- create instances of selections to reflect change deltas

Insert and delete deltas are handled in a similar fashion as they reflect the same situation: one or more event tasks are included in $l(t)$ of one sequence but missing in $l(t)$ of the other sequence. Let t_1 be a sequence where $l(t_1)$ includes the event tasks denoted by an insert or delete delta d and let t_2 be the sequence whose $l(t_2)$ does not include these event tasks. For the delta d, we generate an optional to reflect the delta. If the delta denotes exactly one event task, this event task becomes the single child of the optional. If the delta denotes more than one event tasks, we first generate a new intermediate sequence having the event tasks of d as children and set this sequence as the single child of the optional.

Afterwards, we adapt the flattened instances of both similar sequences. The flattened instances of sequence t_1 are adapted by replacing the instances of the event tasks denoted by d with instances of the new optional. If only one event task instance is replaced, this becomes the single child of the replacing optional instance. If more than one event task instances are replaced at once, we ensure that the replacing optional instance matches the task structure of the corresponding optional including intermediate sequences if any. The flattened instances of t_2 are adapted by integrating instances of the new optional at the position where the event tasks denoted by d are missing. These optional instances do not have children what reflects that the child task of the optional was not performed.

An example for handling an insert and a delete delta, both referring only to one event task, is shown in Figures 9b-d. In Figure 9b, we show the determined deltas and also name the two optionals that will be created to reflect them (Optional *1* for the insert and Optional *2* for the delete delta). Figures 9c and d show, how two exemplifying flattened instances of the similar sequences are adapted by integrating empty optional instances or replacing event task instances. For example, in Figure 9c, the instance of sequence t_1 does not contain an instance of Event Task *b* (indicated also by the insert delta). Hence, we integrate an empty instance of Optional *1* at the respective position between the instances of Event Task *a* and *c*. In addition, we replace the occurrence of Event Task *b* in the instance of t_2 (Figure 9d) with an instance of Optional *1* having the instance of *b* as its child.

To handle a change delta, we generate a selection. This selection gets two children, both representing the appropriate variants defined by the delta. If a variant consists of more than one event task, we again work with intermediate sequences to be able to reflect several subsequent event tasks at once. Afterwards, we adapt all flattened instances of both sequences. We replace each occurrence of the event tasks denoted by the delta with instances of the new selection. The children of the selection instances are set to reflect the executed variant. We again ensure that intermediate sequences, if any, are also correctly reflected in the selection instances. Figures 9b-9d show an example of this approach. Figure 9b shows a change delta for which we generate Selection *1*. Afterwards, we adapt the flattened example instances of t_1 and t_2 as shown in the Figures 9c and 9d. The event task instances *f* in the instance of t_1 and g in the instance of t_2 are both replaced by instances of Selection *1*. In both cases, the selection instance has the replaced event task instance, i.e., the selected variant, as its single child.

4) Iteration and Sequence Detection: After all flattened instances of a sequence pair to be merged are adapted according to the above rules, we reapply our sequence and iteration detection on all adapted flattened instances of both sequences (see substeps four and five in Figure 8). Both substeps are repeated until no more iterations or sequences are detected in the flattened instances. Afterwards, we get a new task structure as replacement for both sequences that now includes selections or optionals to reflect the different task variants. The result of the reapplication of our approach on the adapted flattened instances of Figures 9c and 9d is shown in Figure 9e.

The reapplication of the sequence and iteration detection is always done on at least two flattened task instances (one for each sequence of the merged pair). Furthermore, the flattened instances are adapted in a way so that their structure is the same. Hence, in the last cycle of the sequence and iteration detection, we automatically detect a sequence covering the whole new structure. This new sequence becomes the replacement for both merged sequences.

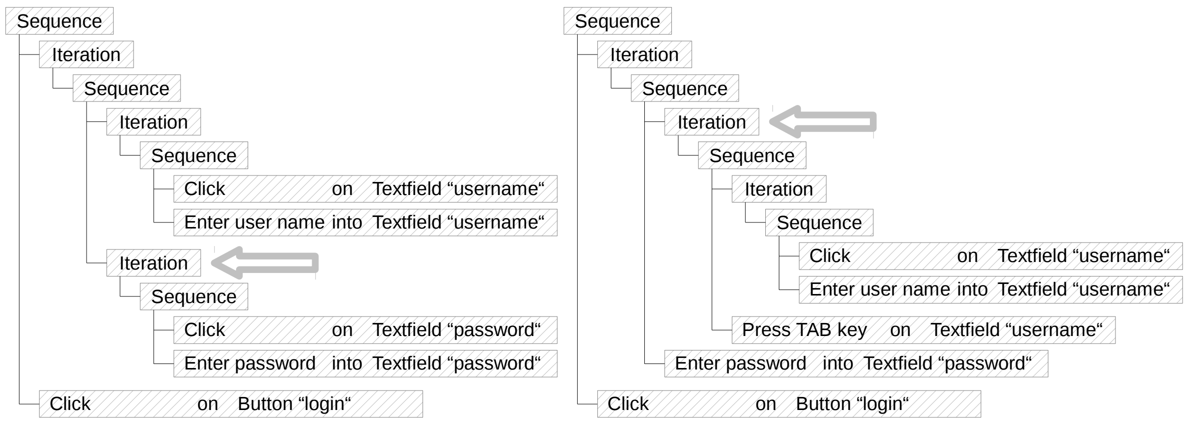
5) Handling of Interleaving Iterations: Although two sequences are similar, they may differ in the possible iterations of event tasks. A typical example is shown in Figure 10. This figure displays two variants of task trees of a login process as they are generated by the alternating iteration and sequence detection of our approach. For navigating from the user name to the password field, the first variant includes the usage of a mouse click on the password field whereas the second variant uses the pressing of the tabulator key. The merging process described in the preceding paragraphs would consider both sequences as similar as they differ only at two of ten event tasks and would merge them. But here, merging must be prevented. The reason is, that the behavior of a click on the password field and the usage of the tabulator key is different with respect to the focus state of the GUI. A repeated click on the password field leaves the focus on the password field. But a repeated usage of the tabulator key will move the focus always to the next element of a form. Hence, depending on the event tasks in both variants of the example the allowed iterations in the variants differ. We call this situation *interleaving iterations*.

Interleaving iterations cannot be merged by our approach. To handle such similar sequences anyway, we determine interleaving iterations before merging. In the example in Figure 10, these are the iterations marked with the grey arrows. If we find interleaving iterations, we change our merging process. First, we calculate a variant of $l(t)$ that we call $l'(t)$. This variant results from a depth first traversal of task t where event tasks are added to $l'(t)$ and interleaving iterations are not traversed but also directly added to $l'(t)$. As a result, $l'(t)$ contains either interleaving iterations or event tasks that are not part of an interleaving iteration. We then apply the diff algorithm $D(l'(t_1), l'(t_2))$ to get an adapted set of deltas. Second, when adapting the flattened instances of two similar sequences, we do not consider fully flattened instances but prevent flattening instances of the interleaving iterations. Furthermore, we consider only the adapted set of deltas when creating optionals and selections. The remaining steps stay the same. Through this, we adapt our overall process to preserve interleaving iterations and, hence, to be correct with respect to allowed or possible repetitions of actions.

6) Reapplication of Similar Sequence Merging: As shown in Figure 8, the detection and merging of similar sequences is repeated until no more similar sequences are found whose similarity level is low enough. This also includes merging of sequences being themselves results of merging or that have results of merging as their subtasks. Due to this, two compared and potentially merged tasks may include selections and optionals that need special considerations when comparing and merging tasks.

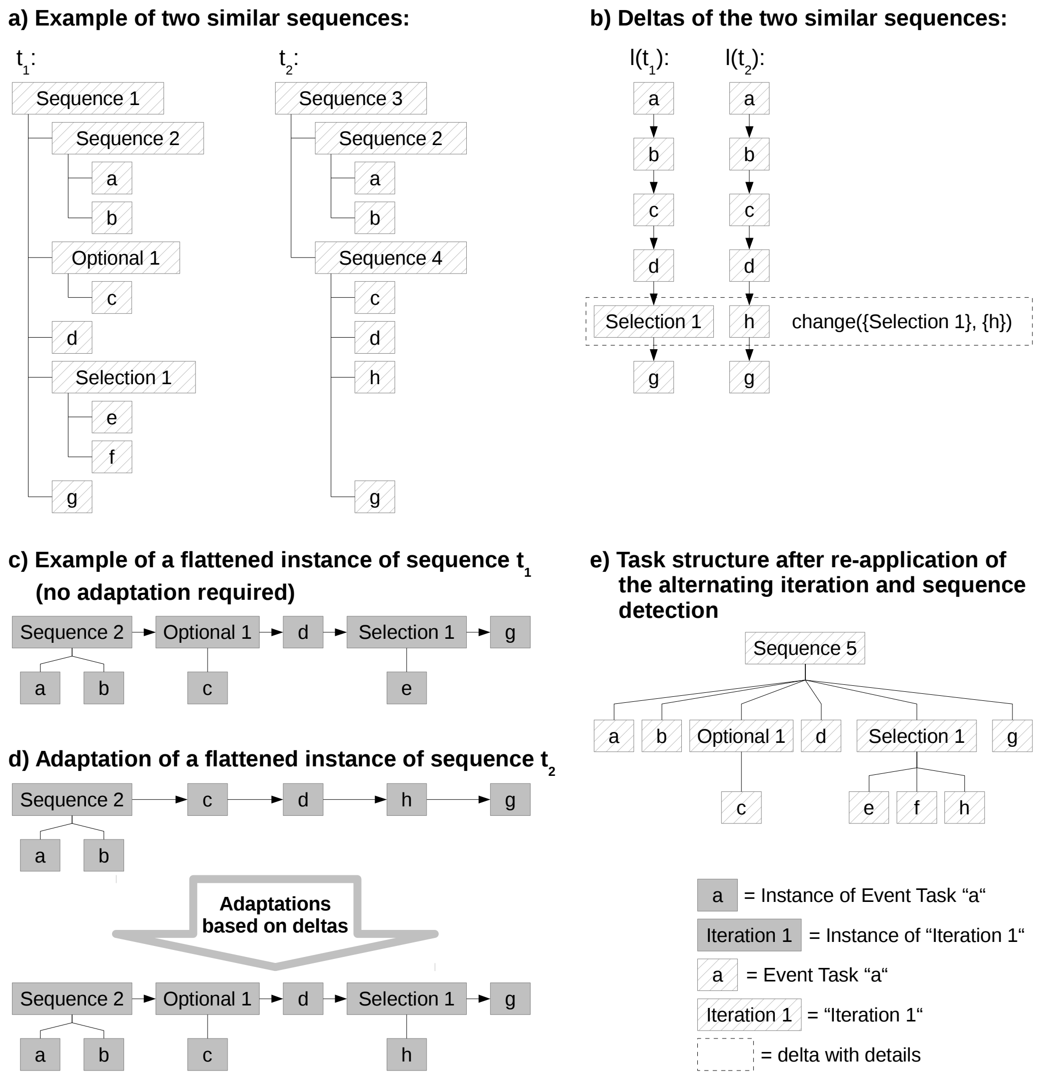
When creating $l(t)$ of a task including an optional, the optional is traversed normally. Hence, $s(t_1, t_2)$ of two tasks that may contain optionals in addition to sequences and iterations is calculated the same way. We make some additional considerations when performing a merging of two task t_1 and t_2. It may be the case, that elements of $l(t_1)$ being equal in $l(t_2)$ may be optional in t_1 but not in t_2 because of a parent optional. An example is shown in Figure 11a. There Event Task c is optional in task t_1 but mandatory in t_2. But this does not show up as a delta between $l(t_1)$ and $l(t_2)$ as seen in Figure 11b. To anyway preserve such optionals, we first identify those elements of $l(t_1)$ that have an optional as parent. Instances of these optionals are not flattened when creating the flattened instances of t_1. Afterwards, we ensure that all elements of $l(t_2)$ have an optional as parent if the corresponding elements of $l(t_1)$ also have an optional as its parent. If required, an optional is introduced. When creating the flattened instances of t_2, instances of these optionals also remain unflattened. As a result, all flattened instances of both tasks t_1 and t_2 will contain event task instances and optional instances. This ensures that the optionals contained in t_1 are preserved. Figures 11c and 11d show how the instances of the sequences t_1 and t_2 are flattened by preserving (Figure 11c) and respectively introducing (Figure 11d) an optional for Event Task c.

Selections require a more complex handling. If a task contains a selection, this selection can not be traversed when creating $l(t)$. Hence, $l(t)$ of a task containing a selection includes event tasks and selections. An example for this is also



Sequence
Iteration
Sequence
Iteration
Sequence
Click on Textfield “username“
Enter user name into Textfield “username“
Iteration
Sequence
Click on Textfield “password“
Enter password into Textfield “password“
Click on Button “login“
Sequence
Iteration
Sequence
Iteration
Sequence
Iteration
Sequence
Click on Textfield “username“
Enter user name into Textfield “username“
Press TAB key on Textfield “username“
Enter password into Textfield “password“
Click on Button “login“

Figure 10. Example of two similar tasks with interleaving iterations



a) Example of two similar sequences:
t1:
Sequence 1
Sequence 2
a
b
Optional 1
c
d
Selection 1
e
f
g
t2:
Sequence 3
Sequence 2
a
b
Sequence 4
c
d
h
g
b) Deltas of the two similar sequences:
l(t1):
l(t2):
a
b
c
d
Selection 1
h
change({Selection 1}, {h})
g
c) Example of a flattened instance of sequence t1 (no adaptation required)
Sequence 2
Optional 1
d
Selection 1
g
a
b
c
e
d) Adaptation of a flattened instance of sequence t2
Sequence 2
c
d
h
g
a
b
Adaptations based on deltas
Sequence 2
Optional 1
d
Selection 1
g
a
b
c
h
e) Task structure after re-application of the alternating iteration and sequence detection
Sequence 5
a
b
Optional 1
d
Selection 1
g
c
e
f
h
a = Instance of Event Task “a“
Iteration 1 = Instance of “Iteration 1“
a = Event Task “a“
Iteration 1 = “Iteration 1“
= delta with details

Figure 11. Example of merging two similar tasks containing optionals and selections

shown in Figure 11. There, Sequence t_1 contains a selection that is not flattened and hence shows up in $l(t_1)$.

Although a selection is not flattened when creating $l(t)$, it may represent several event tasks at once. Thus, we adapt the calculation of $s(t_1, t_2)$ to still correctly represent the ratio of different event tasks for the comparison of t_1 and t_2 even if one of them contains a selection. For this, we calculate the average number of event tasks $e(p)$ covered by each selection $p \in l(t_1) \cup l(t_2)$. This is done by creating $l(c)$ for each child c of a selection and then calculating the average length of these lists. Furthermore, if a delta d includes one or more selections, then $e'(d)$ is the sum of the average event tasks covered by the selections and the remaining number of event tasks belonging to the delta:

$$e'(d) = \sum_{p \in d} e(p) + e(d) \tag{3}$$

Finally, the calculation of $s(t_1, t_2)$ is adapted to respect the average number of event tasks covered by selections as follows:

$$s(t_1, t_2) = \frac{o(t_1, t_2) + \sum_{i=1}^{n} (e'(d_i) - o(d_i))}{|l(t_1)| + |l(t_2)| + \sum_{p \in l(t_1) \cup l(t_2)} (e(p) - 1)} \tag{4}$$

For merging two sequences, where one contains one or more selections, we do not flatten selection instances. This is to preserve the selection instance and its selected variant. Furthermore, if a selection spans a change delta completely, we adapt it to include the new variant in its children. An example for this is shown in Figure 11a. There, t_1 includes a selection. When flattening an instance of t_1 (Figure 11c) the instance of the selection is preserved and not flattened. When adapting the flattened instance of the similar task t_2, the selection is extended with a further variant (Event Task h) as the delta between t_1 and t_2 is fully spanned by the selection. Hence, the flattened instance of t_2 is also adapted by replacing the instance of Event Task h with an instance of the selection having h as the selected variant.

A further consideration in our merging process is the preservation of subtasks being either fully covered by a delta or representing common parts of two similar tasks. An example is shown in Figure 11a. Both sequences start with the sub task Sequence 2. When merging t_1 and t_2, this subtask and all its instances can and should be preserved. For this, when flattening the instances of two similar tasks, we do not flatten instances of subtasks being either common for both tasks or being fully covered by a delta. In the example in Figure 11c and d, the instances of Sequence 2 stay unflattened.

For each detected optional and selection as well as for intermediate sequences, we ensure that the same task is created only once. For example, if during several merges an optional of a specific event task must be created, the optional is created only during the first merge and then reused in the other merges. This also requires to detect if a generated task matches the requirements of a new task to be created.

F. *Usability Evaluation*

We utilize the generated task trees for automated usability evaluations. For this, we consider violations of generally accepted usability heuristics (e.g., as provided in [12]) and define patterns for their reflection in task trees. We then filter our task trees for these patterns and reason on potential usability defects. This is possible, as the generated task trees represent effective user behavior. For example, we analyze typical action combinations or navigation patterns users perform on a website and detect if their efficiency could be improved. The details of this approach can be found in [6].

III. Proof-of-concept Implementation

To show that our method is feasible, we implemented it based on the tool suite for Automatic Quality Engineering of Event-driven Software (AutoQUEST) [13]. The AutoQUEST platform provides diverse methods for assessing the quality of software. AutoQUEST's internal algorithms operate on abstract events, which makes AutoQUEST independent of the platform of an assessed software. AutoQUEST's modular architecture allows the extension with modules to support algorithms for quality assurance, as well as feeding AutoQUEST with events of a yet unsupported software platform. In the following, we describe how we utilized and extended AutoQUEST to implement our method.

A. *User Interaction Tracing*

AutoQUEST provides basic functionality for tracing user actions on different platforms including websites. For this, it uses techniques from GUI testing, e.g., for capture/replay testing [14]. For monitoring a website only a JavaScript needs to be added to each of the pages of the website. In modern content management systems, this can be configured centrally and easily. The JavaScript is served by a monitoring server shipped with AutoQUEST. It automatically records events caused by user actions. After a specific amount of events is recorded, or if the user switches the page, the script sends the events to the AutoQUEST server that stores them into log files. Using a dedicated parser, these log files can then be fed into AutoQUEST for further processing.

An excerpt of a trace of AutoQUESTs website monitor showing a mouse click and a scroll event on a web page is shown in Figure 12. Both events denote their respective type, a timestamp, and meta information like the coordinates in the click event. Furthermore, both events refer to a target, i.e., the element of the webpage, on which the event was observed. The identifiers of the targets can be resolved through other information stored in the log file, as well.

B. *Task Tree Generation*

For our proof of concept implementation, we extended AutoQUEST with capabilities to generate task trees based on traces. The implementation follows the overall process described in Section II. The implementations of the data preparation and the iteration detection are straightforward and, therefore, not described in more detail.

```
<event type="onclick">
 <param name="X" value="87"/>
 <param name="Y" value="213"/>
 <param name="target" value="id1"/>
 <param name="timestamp" value="1375177632056"/>
</event>
<event type="onscroll">
 <param name="scrollX" value="-1"/>
 <param name="scrollY" value="-1"/>
 <param name="target" value="id2"/>
 <param name="timestamp" value="1375177632900"/>
</event>
```

Figure 12. Example for a trace recorded with AutoQUEST's HTML monitor

1) Sequence Detection Implementation: For identifying and counting subsequences occurring several times, we reused and extended a data structure provided with AutoQUEST called trie [8]. A trie in AutoQUEST is a tree structure used for representing occurrences of subsequences in a sequence. In our case, we use the trie for representing subsequences of tasks in the ordered list of tasks considered for the next sequence detection. An example for a trie is shown in Figure 13.

Each node in a trie represents a task subsequence. The length of the represented subsequence is equal to the distance of the node to the root node of the trie. The root node of the trie represents the empty subsequence. The children of the root node (in Figure 13 all nodes on Level 1) represent the subsequences of length 1 occurring in the trace, i.e., all different tasks. The grand children of the root node (in Figure 13 all nodes on Level 2) represent the subsequences of length two as their distance to the root node is two, etc. The subsequence represented by a node can be determined by following the path through the trie starting from the root node and ending at the respective node. The length of the longest subsequence represented through a node in the trie is defined as the depth of the trie. The depth of the trie in Figure 13 is three.

Each node in a trie is assigned a counter. This counter defines the number of occurrences of the subsequence represented by the node. The counter of the root node is ignored. The example trie in Figure 13 represents the event tasks for the trace of Figure 4. The trie shows that the event task of clicking on the user name text field occurs twice and that both times it is succeeded by entering some text, i.e., a user name, into the text field. The event of clicking the login button is not succeeded by any other event task.

We calculate a trie each time a sequence detection on the ordered list of tasks is done. Based on the trie, we are able to identify the longest subsequence of tasks with a minimal length of two occurring most often. The number of occurrences is determined through the counts assigned to each node in the trie. The length of the subsequence is determined by the distance of the trie node representing the most occurring subsequence to the root node of the trie.

If the length of the identified subsequence is identical to the depth of the trie, we cannot decide if there is a longer subsequence with the same count. We, therefore, increase the depth of the trie until the depth is larger than the length of the longest subsequence occurring most often. In Figure 13, the longest subsequence occurring most often is clicking on the user name text field and entering a user name. This subsequence occurs twice and there is no other subsequence of the same or a longer length occurring more often. Therefore, all occurrences of this subsequence in the ordered list of tasks is replaced through a task node of type sequence.

2) Comparison of Tasks: An important challenge in our implementation was the comparison of tasks. Tasks need to be compared very often either for compiling the trie or for detecting iterations. For an effective task generation, some tasks must be considered equal although they are different. An example is a task and an iteration of this task. Both must be considered identical if the iteration is executed only once. Another example is shown in Figure 13. The represented trie contains nodes for the event tasks representing the entering of text into the user name text field. Although different text is entered in the respective events, the respective event tasks need to be considered identical for a correct trie calculation. Therefore, we implemented a mechanism to be able to perform complex task comparisons. In addition to other comparisons, it is able to compare a task A with an iteration of a task B and considers them as equal if task A is equal to task B.

3) Merging of Similar Sequences: Due to the large number of sequence comparisons required for detecting similar sequences, we implemented the comparisons to be executed in parallel. For this, we schedule several threads each performing a bucket of all required comparisons. The number of threads executed in parallel can be configured and should match the number of available cores on a machine. Each thread searches for those sequence pairs in its buckets that have the lowest similarity level and returns them. Afterwards, the results of all threads are joined and again the most similar sequences are filtered out.

Furthermore, we implemented some checks to ensure that the merging was correct. For example, we ensure that the original flattened instances of two merged tasks are identical to the flattened instances of their replacement task. Additionally, we performed random manual checks of what is merged and if the merge results are correct.

IV. Case Studies

For the validation of our approach, we performed two case studies. With these case studies, we intended to show that our approach is feasible and able to generate task trees based on recorded user actions. For this, we first recorded user actions on two websites. Then we used our approach, i.e., its implementation, to generate task trees. Finally, we evaluated the correctness of the detected task trees through manual inspection and comparison with the structure and available user actions of the website and its pages. However, due to the partially large number of generated tasks, this inspection was only done for a subset of tasks.

For the first case study, we traced the interaction of users of our research website [15]. This website provides three major information categories to its users: information about our research group members, information about our research and corresponding projects, as well as information about the

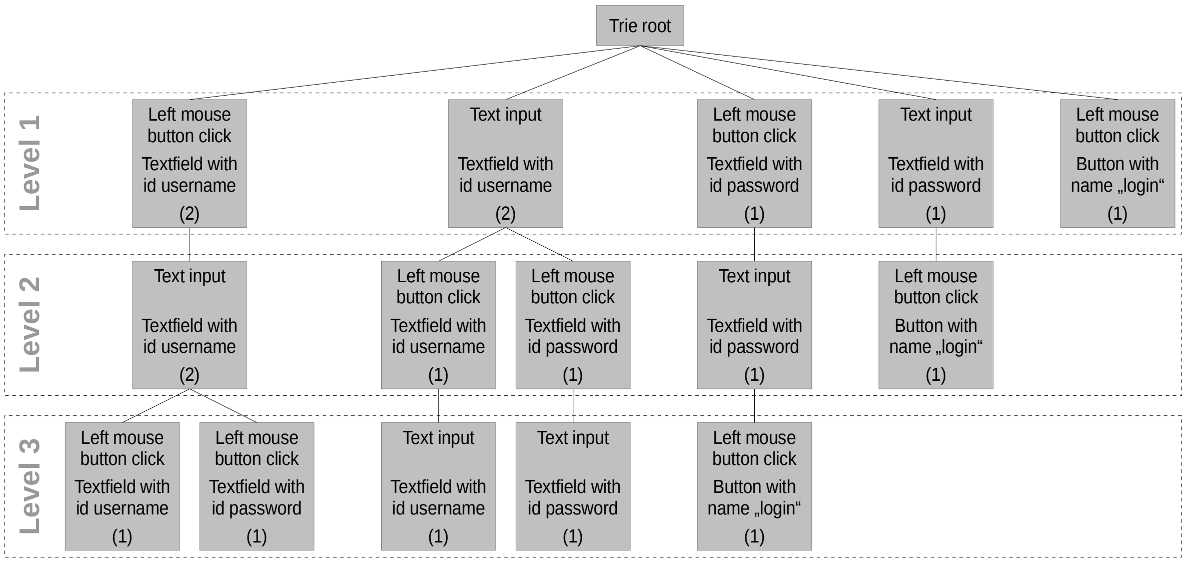
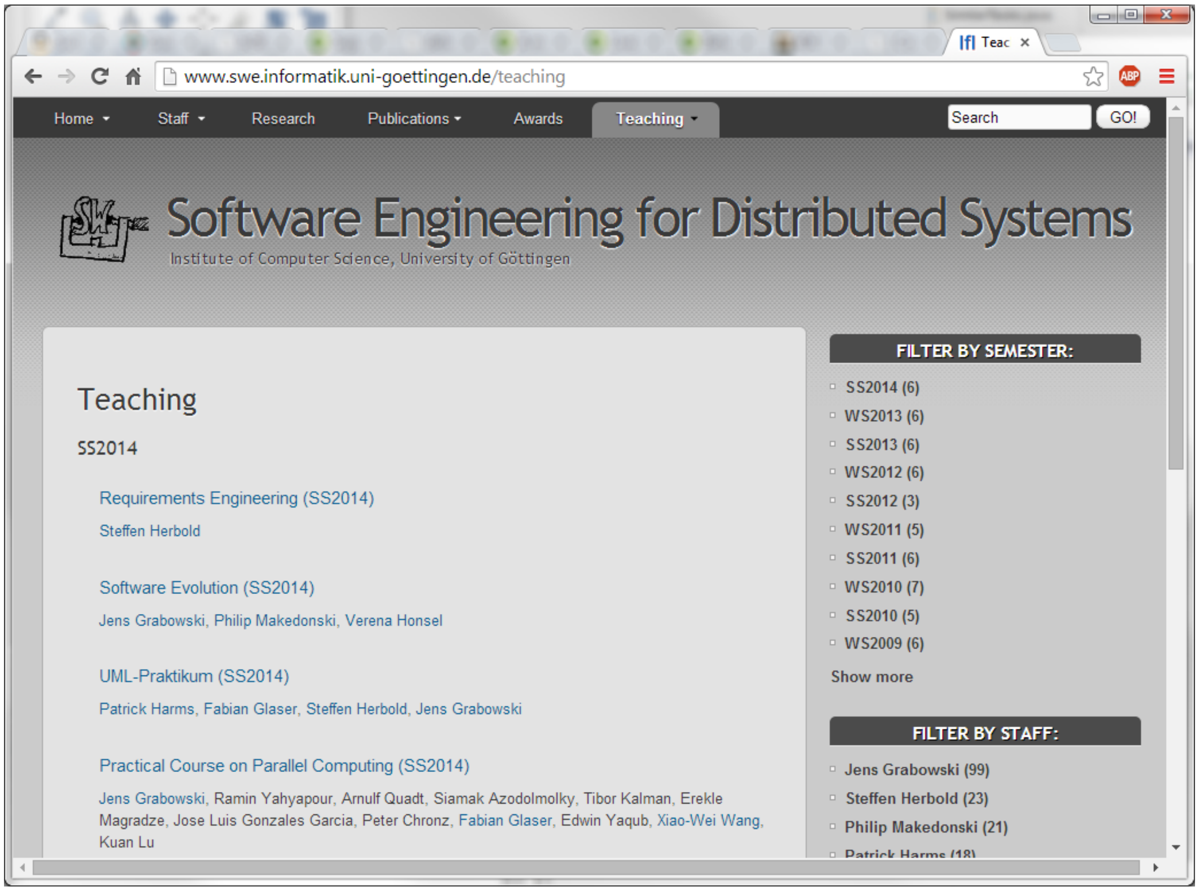


Figure 13. Trie generated based on the trace in Figure 4


www.swe.informatik.uni-goettingen.de/teaching
Home
Staff
Research
Publications
Awards
Teaching
Search
GO!
Software Engineering for Distributed Systems
Institute of Computer Science, University of Göttingen

Teaching

SS2014

Requirements Engineering (SS2014)
Steffen Herbold

Software Evolution (SS2014)
Jens Grabowski, Philip Makedonski, Verena Honsel

UML-Praktikum (SS2014)
Patrick Harms, Fabian Glaser, Steffen Herbold, Jens Grabowski

Practical Course on Parallel Computing (SS2014)
Jens Grabowski, Ramin Yahyapour, Arnulf Quadt, Siamak Azodolmolky, Tibor Kalman, Erekle Magradze, Jose Luis Gonzales Garcia, Peter Chronz, Fabian Glaser, Edwin Yaqub, Xiao-Wei Wang, Kuan Lu

FILTER BY SEMESTER:
- SS2014 (6)
- WS2013 (6)
- SS2013 (6)
- WS2012 (6)
- SS2012 (3)
- WS2011 (5)
- SS2011 (6)
- WS2010 (7)
- SS2010 (5)
- WS2009 (6)

Show more

FILTER BY STAFF:
- Jens Grabowski (99)
- Steffen Herbold (23)
- Philip Makedonski (21)
- Patrick Harms (18)

Figure 14. Screenshot of the Research Website of the first case study

courses we offer to students at our institute. A screenshot of the page listing the offered courses is shown in Figure 14.

We integrated the HTML monitor of AutoQUEST in our content management system. We then recorded interactions of

TABLE I. RECORDED AND CONSIDERED EVENTS IN THE TWO MAJOR CASE STUDIES

	Case Study 1 Website of Research Group	**Case Study 2** Application Portal
Start of Recording	30 June 2013	25 October 2013
End of Recording	4 March 2014	7 March 2014
Recorded Users	1,356	555
Recorded User Sessions	8,113	4,129
Considered User Sessions	6,587	3,635
Recorded Events	63,127	350,368
Relevant Events	38,070	306,568
Double Clicks	741	6,437
Focus Changes	9,950	89,825
Considered Events	27,379	210,306
Different Events	1,202	1,897
Generated Tasks Without Merging	1,847	10,634
Sequences	1,431	9,530
Iterations	416	1,104
Generated Tasks With Merging	1,842	10,663
Sequences	1,419	9,361
Iterations	416	1,133
Selections	3	81
Optionals	4	88

1,356 users over a period of 8 months. Afterwards, we fed the gathered traces containing more than 63,000 events into AutoQUEST and generated over 1,800 task trees based on this. Further details of the case study are listed in the middle column of Table I.

This case study showed that the task tree generation was feasible in general. The generated task trees represented user behavior occurring several times. As an example, several users opened the initial web page and navigated to our teaching page (shown in Figure 14). From there, they navigated to the information about a specific course by clicking on one of the respective links.

In the first case study, we set s_{max} to 0.33, i.e., we considered two sequences as similar if less than or equal to one third of the event task lists of both sequences are different (including scrolls). We chose this border to reflect that we expect at least twice as many commonalities than differences between sequences to consider them as similar. Using this parameter, the merging of tasks in the first case study found 12 similar tasks and merged them resulting in three selections and four optionals. One optional was detected and reused in 6 merges, one selection in two merges. The merged tasks were executed quite seldom and covered at most 262 events per similar task pair. The optional that was integrated most often represents an optionality of scrolling.

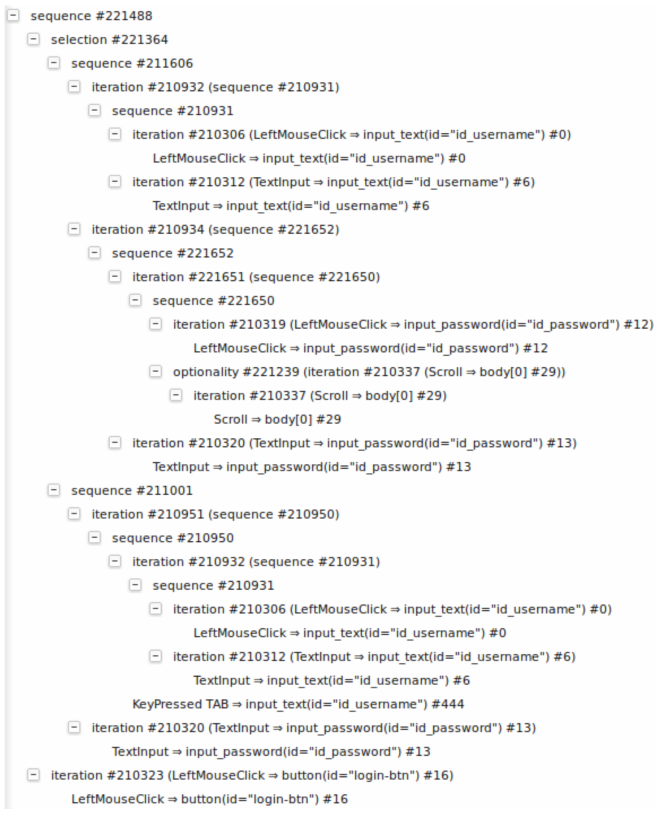
The first case study revealed that our mechanism must be careful with respect to privacy protection. Our research website includes a log-in mechanism for being able to change its content. The first version of the tracing mechanism also traced user names and passwords of all users that logged in on the website. As this was a severe security issue, we adapted the tracing mechanism to ignore password fields in general. Furthermore, a website can be instrumented in a way, so that contents of selected text fields, e.g., fields for entering a user name, are not traced anymore.

In our second case study, we traced the users of an application portal of our university over a period of 4 months. This case study traced over 550 users producing more than 350,000 events resulting in 10,663 generated task trees. The details for this case study are listed in the right column of Table I. As in the first case study, we set s_{max} to 0.33.

When feeding the data of the second case study into AutoQUEST, we initially observed performance problems of our approach. Especially, the large number of distinct events caused the creation of a large trie for sequence detection. We, therefore, implemented several optimizations. For example, click events on the same button but with different coordinates are now treated as the same event task. However, click events on other website elements are still considered different, if their coordinates differ. Furthermore, we made intensive use of the mapping of common page elements and thereby reduced the number of different event tasks. This was done as on some pages of the portal, tables were used to represent content of similar meaning (e.g., a list of the application data entered by a user). For each row in such a table, there were buttons to edit the content. Although each of these buttons refer to different content to be edited, the basic semantic of these buttons is considered the same: edit the content belonging to the row. Hence, we considered these buttons as common page elements.

The second case study showed that our approach is able to correctly identify effective user behavior. The application portal also provides a login mechanism. Our task tree generation created several different task trees for the login process of users. One of them showed the behavior of those users using the mouse to set the focus on the password field after having entered the user name. Another task tree showed the usage of the tabulator key instead. The merging process correctly identified these two tasks as similar and merged them considering the interleaving iterations. A visualization of the merge result as displayed by AutoQUEST is shown in Figure 15. The example shows, that many iterations are generated in the task tree. This is due to the fact, that some users corrected the entered data several times. Furthermore, if the users entered wrong credentials, the website returned to the same page and the users started the login process again. The selection resulting from the merge is very large due to the handling of interleaving iterations.

In the second case study, the merging process identified more similar task than in the first case study and performed 224 mergings. This resulted in 81 selections and 88 optionals. The selection that covered most recorded events was a selection between entering a date into five different text fields and was mainly integrated into tasks representing the usage of a date chooser. The optional that covered most recorded events was also here an optional of scrolling. Due to the merging, the overall number of tasks increased, as new selections and optionals were introduced but fewer other tasks (sequences) where discarded. Furthermore, more iterations were generated. This is because two similar tasks may have been executed subsequently. Through the merging, they were afterwards



```
sequence #221488
  selection #221364
    sequence #211606
      iteration #210932 (sequence #210931)
        sequence #210931
          iteration #210306 (LeftMouseClick ⇒ input_text(id="id_username") #0)
            LeftMouseClick ⇒ input_text(id="id_username") #0
          iteration #210312 (TextInput ⇒ input_text(id="id_username") #6)
            TextInput ⇒ input_text(id="id_username") #6
      iteration #210934 (sequence #221652)
        sequence #221652
          iteration #221651 (sequence #221650)
            sequence #221650
              iteration #210319 (LeftMouseClick ⇒ input_password(id="id_password") #12)
                LeftMouseClick ⇒ input_password(id="id_password") #12
              optionality #221239 (iteration #210337 (Scroll ⇒ body[0] #29))
                iteration #210337 (Scroll ⇒ body[0] #29)
                  Scroll ⇒ body[0] #29
          iteration #210320 (TextInput ⇒ input_password(id="id_password") #13)
            TextInput ⇒ input_password(id="id_password") #13
    sequence #211001
      iteration #210951 (sequence #210950)
        sequence #210950
          iteration #210932 (sequence #210931)
            sequence #210931
              iteration #210306 (LeftMouseClick ⇒ input_text(id="id_username") #0)
                LeftMouseClick ⇒ input_text(id="id_username") #0
              iteration #210312 (TextInput ⇒ input_text(id="id_username") #6)
                TextInput ⇒ input_text(id="id_username") #6
          KeyPressed TAB ⇒ input_text(id="id_username") #444
      iteration #210320 (TextInput ⇒ input_password(id="id_password") #13)
        TextInput ⇒ input_password(id="id_password") #13
  iteration #210323 (LeftMouseClick ⇒ button(id="login-btn") #16)
    LeftMouseClick ⇒ button(id="login-btn") #16
```

Figure 15. Task tree generated in the context of the second case study

considered the same task. Hence, their subsequent execution was correctly identified as an iteration of the merged task.

V. Discussion

The generated task trees represent the effective user behavior. This is important to analyze the usage of a monitored website, e.g., with respect to usability. Our approach is also able to identify distinct ways of executing semantically equal tasks and merge them into a single task.

At each repetition, the detection of sequences chooses the longest subsequence occurring most often and replaces it as described. This heuristic prefers shorter sequences as the count of a subsequence decreases with an increasing length. The resulting task trees are, therefore, deeply structured. Hence, it would be better to apply a more sophisticated heuristic such as selecting a subsequence occurring more seldom but being much longer.

Currently, we identify tasks that are executed only seldom. For example, we generate sequences for event combinations that happened only twice during user tracing. In the future, we plan to perform a filtering so that tasks must have a minimum amount of covered recorded events to be considered as tasks. This can be seen as a measure for the evidence of a task to be really a common task for all users.

For user interactions or tasks there may be pre or post conditions. For example, an iteration can be repeated a minimum or maximum amount of times. Our approach is not able to detect these conditions. Therefore, the task tree structures that we generate do not include notations for conditions.

The merging process allows to have in the end less tasks describing the semantically same task with slightly different execution variants. However, the merged tasks and their parent tasks may not be fully correct anymore with respect to the possible action sequences they represent. For example, in the second case study, the process merged several similar tasks of using a date chooser. All these tasks were identical except for the event task that finally entered the selected date into the respective text field. The merging process created a selection of these event tasks and merged several date chooser usages into a merged task. The last element of this merged task was a selection of entering a date into several available date fields on different pages of the website. Unfortunately, the merged task was a child of other parent tasks, e.g., of a parent task that was executed only on a specific page of the website. Hence, because the merged task also represented usages of the date chooser on other pages of the website, not all of its execution variants were valid in the context of this parent task. But for analyzing the usage of the date chooser, this merging was very helpful.

As a result, also invalid parent tasks were created through merging. This also applies for all parent hierarchies of a merged task. To check, if a parent task becomes invalid trough merging, a manual inspection was required. In the first case study, we did not observe this issue. In the second case study, we identified 8 of the 81 selections to cause invalid parent tasks. All were related to the date chooser usage. The cause for this issue is the consideration of common page elements. All date choosers on all different pages were considered as common page elements. But although being positioned at the same location in the DOMs of the different pages, they were semantically related to a specific page and a specific date field and, hence, should not be considered common. Therefore, this issue is not caused by the merging process itself but by the data preparation.

The merging process is based on the detection of similar tasks using Myers diff algorithm. In this work, we have not evaluated if the application of other diff algorithms would reveal other merging results. Hence, the results of the merging process may depend on the used diff algorithm. Furthermore, we set s_{max} in both case studies to a fixed value. We have not evaluated how a change on s_{max} may affect the merge result.

In some situations, our merging process does not yet fully correctly merge execution variants. For example, our approach created several selections, of which at least one child was a task and another child contained the task too as some embedded child. The simplest example is a selection of a task and an optional of the same task. Hence, some generated task structures are still more complex than they could be and need to be further refined.

VI. Related Work

In this section, we refer to related work. We start with similar work on recording user actions. Then, we consider different approaches for task modeling and compare them with our work. Finally, we compare our approach with other attempts to generate task trees.

A. Recording of user actions

Nowadays, the idea of tracing user actions on websites is often applied in the context of web analytics, e.g., with Google Analytics [16] and Piwik [17]. Furthermore, there exist several tools that can be used to trace software based on other platforms than HTML. In contrast to our approach, the level of detail of the information recorded by existing tracing mechanisms varies and is often rather low. For example, Piwik does not record individual clicks.

To get more detailed recordings, other approaches, e.g, the one used by WebRemUsine [5], require Java applets or other mechanisms to store the recorded user interactions on the client and send them subsequently to the server. UsaProxy [18] provides an Hyper-Text Transfer Protocol (HTTP) proxy that is located between a web server and its clients. The proxy adapts each HTML document requested by a client and inserts a reference to a Javascript. This script automatically records detailed user actions at client side and sends them via Asynchronous JavaScript and XML (AJAX) [19] to the proxy that in turn stores them. Our approach for recording user actions on websites also utilizes an AJAX approach but without using a proxy. Instead, our Javascript is integrated in each website using mechanism of the content management system of a website. This has the advantage, that we can distinguish between monitored and unmonitored parts of a website. Furthermore, instead of using a standard way for instrumenting all pages of a website as done by UsaProxy, our approach allows to consider website specific technical challenges when adding the Javascript to the pages.

B. Task Modeling

Task models are used to describe the actions a person has to perform to reach a specified goal. In the context of website usage, they allow to define, which and how website elements are to be used to accomplish one of the tasks the website was developed for [5]. Task models usually cover task decomposition, task flow specification, object modeling, and task world modeling [20]. Our approach is restricted to task decomposition and task flow specification. Task models can be either used for aiding design, validating design decisions, or, in the most formal way, generating user interfaces [20]. Our task trees aid design and are restricted to summative validation.

Van Welie et al. [20] developed a common ontology for task models as a basis for a harmonized comparison of different task modeling approaches. The ontology covers concepts and their relationships that are typically used in task modeling. The concept that is covered by our approach is *Task* with its more concrete variants *basic task* and *user action* (identical to the term *action* in our approach). Van Welie et al. define a basic task as "... a task for which a system provides a single function. Usually[,] basic tasks are further decomposed into user actions and system operations". The task structures that we generate in our approach are mainly on the level of basic tasks. However, the root nodes of our generated task trees are similar to Van Welies unit tasks that they consider "... as the simplest task that a user really wants to perform". We do not cover other concepts of Van Welies ontology. The relationships defined in the ontology that are also covered by our approach are *subtask* and *triggers*. The subtask-relationship defines the child tasks or actions of a task. The trigger-relationship defines the order of task execution. The trigger-relationship can have three different types of which we cover only *NEXT* and *OR*. NEXT indicates, which task is executed next what is covered by our temporal relationship sequence. OR indicates that there is a choice between several next tasks. This is covered in our approach by selection (simple choice), iterations (repeat or go on with next task/action), and optional (perform or skip task/action). Due to this possible mapping to Van Welies ontology, our task trees should be transformable into other task tree notations that can be mapped to the ontology, as well. But this is scheduled for future work.

Task trees are one possible variant for modeling tasks. The concept of task trees is applied, e.g., in Goals, Operators, Methods, and Selection Rules (GOMS) [21], TaskMODL [22], and ConcurTaskTrees [23] [7]. If the nodes of task trees define the temporal relationships for their children, they have the drawback that intermediate nodes may be required to fully specify the task flow [20]. An alternative for task modeling are workflow representations that do not have this drawback but require a time axis. In our approach, we use the basic concept of task trees, but apply it in a simplified manner.

C. Task Tree Generation

There have been several attempts to generate task trees automatically. For example, the Convenient, Rapid, Interactive Tool for Integrating Quick Usability Evaluations (CRITIQUE) [24] creates GOMS models based on recorded traces. A similar approach is proposed by John et al. [25]. ReverseAllUIs [2] generates task trees based on models of the GUI. The resulting task trees represent all available interactions a user can perform. In contrast to our work, these approaches do not generate task trees that represent the effective behavior of the users, but only a simplified or complete task tree of a website.

A further attempt to identify reoccurring user behavior is programming by example. Here, user actions are recorded to determine reoccurring action sequences. The system then offers the user an automation of the identified action sequence. An example of this work can be found in [26]. These approaches only attempt to locally optimize the usability, whereas we adopt a global view on the system.

Generating task trees for user actions is similar to the inference of a grammar for a language. The user actions are the words of a language that the user "speaks" to the software. The task tree is the grammar defining the language structure. However, current approaches for grammatical inference require the identification of sentences of the language before the derivation of the grammar [27]. For example, during one user session a user may execute several tasks or interrupt a task execution. This would lead to several sentences following each other or incomplete sentences in a user session. Hence, to apply grammatical inference, it would be required to mark those actions in a recorded user session that together form a sentence and to drop incomplete sentences. This would practically not be feasible in the case of a large set of recorded user sessions. Our approach is capable of handling large amounts of recorded actions without requiring a manual marking of correct task executions.

VII. Summary and Outlook

In this paper, we described a method for generating task trees based on tracing user interactions. First, sequences and iterations of user actions are identified. Then, semantically similar sequences are merged. We implemented this method for websites and performed two case studies to validate its feasibility.

In our future work, we will improve and extend the task tree generation. We especially focus on filtering tasks based on a minimum number of covered recorded events. We will ensure that parent tasks will not become invalid due to the extension with execution variants by merging child tasks. Furthermore, we plan to implement a better heuristic for detecting more intuitive subsequences, a flattening algorithm for reducing the complexity of the generated task trees, and an export of our task trees into a format used by other tools, e.g., into the format utilized by the ConcurTaskTrees Environment [28]. In addition, we improve the existing AutoQUEST plug-ins and implement plug-ins for further platforms, e.g., for operating systems with a focus on touch-based interaction. Finally, we improve the automated usability evaluation based on the generated task trees and perform comparisons of the usability evaluation results depending on if the evaluation is based on merged or unmerged tasks.

Acknowledgment

This work was partially done in the context of the project MIDAS (Model and Inference Driven - Automated testing of Services Architectures), funded by the European Commission, project number 318786.

References

[1] P. Harms, S. Herbold, and J. Grabowski, "Trace-based task tree generation," in Proceedings of the ACHI 2014, The Seventh International Conference on Advances in Computer-Human Interactions. ThinkMind, 2014, pp. 337–342.

[2] R. Bandelloni, F. Paternò, and C. Santoro, "Engineering interactive systems," J. Gulliksen, M. B. Harning, P. Palanque, G. C. Veer, and J. Wesson, Eds. Berlin, Heidelberg: Springer-Verlag, 2008, ch. Reverse Engineering Cross-Modal User Interfaces for Ubiquitous Environments, pp. 285–302.

[3] F. Paternò, "Model-based tools for pervasive usability," Interacting with Computers, vol. 17, no. 3, 2005, pp. 291–315.

[4] L. Paganelli and F. Paternò, "Tools for remote usability evaluation of web applications through browser logs and task models," Behavior Research Methods, vol. 35, 2003, pp. 369–378.

[5] F. Paternò, "Tools for remote web usability evaluation," in HCI International 2003. Proceedings of the 10th International Conference on Human-Computer Interaction. Vol.1, vol. 1. Erlbaum, 2003, pp. 828–832.

[6] P. Harms and J. Grabowski, "Usage-based automatic detection of usability smells," in Proceedings of the HCSE 2014, The Fifth International Conference on Human-Centered Software Engineering, 2014.

[7] F. Paternò, "ConcurTaskTrees : An engineered approach to model-based design of interactive systems," The Handbook of Analysis for HumanComputer Interaction, 1999, pp. 1–18.

[8] S. Herbold, "Usage-based Testing of Event-driven Software," Ph.D. dissertation, University Göttingen, June 2012 (electronically published on http://webdoc.sub.gwdg.de/diss/2012/herbold/ [retrieved: 1, 2014]), 2012.

[9] R. J. Jacob, "User interface," in Encyclopedia of Computer Science, ser. Encyclopedia of Computer Science, A. Ralston, E. Reilly, and D. Hemmendinger, Eds. Nature Publishing Group London, 2000, pp. 1821–1826.

[10] J. Foley, Computer Graphics: Principles and Practice, ser. Systems Programming Series. Addison-Wesley, 1996.

[11] E. Myers, "Ano(nd) difference algorithm and its variations," Algorithmica, vol. 1, no. 1-4, 1986, pp. 251–266. [Online]. Available: http://dx.doi.org/10.1007/BF01840446

[12] U.S. Department of Health & Human Services. Usability.gov - improving the user experience - guidelines. [Online]. Available: http://guidelines.usability.gov/ [retrieved: 11, 2014] (2013)

[13] S. Herbold and P. Harms, "AutoQUEST - Automated Quality Engineering of Event-driven Software," March 2013.

[14] J. H. Hicinbothom and W. W. Zachary, "A Tool for Automatically Generating Transcripts of Human-Computer Interaction," in Human Factors and Ergonomics Society 37th Annual Meeting, vol. 2 of Special Sessions, 1993, p. 1042.

[15] Software Engineering for Distributed Systems Group. Software Engineering for Distributed Systems. [Online]. Available: http://www.swe.informatik.uni-goettingen.de/ [retrieved: 7, 2014] (2014)

[16] Google. Google analytics. [Online]. Available: http://www.google.com/analytics/ [retrieved: 11, 2014] (2014)

[17] Piwik.org. Piwik - real time analytics reports for your websites. [Online]. Available: http://de.piwik.org/ [retrieved: 11, 2014] (2014)

[18] R. Atterer, "Usability tool support for model-based web development," dissertation, Oktober 2008. [Online]. Available: http://nbn-resolving.de/urn:nbn:de:bvb:19-92963

[19] J. J. Garrett, "Ajax: A New Approach to Web Applications," 2005, adaptive Path LLC, http://www.adaptivepath.com/publications/essays/archives/000385.php.

[20] M. Van Welie, G. C. Van Der Veer, and A. Eliëns, "An ontology for task world models," in Proceedings of DSV-IS98, Abingdon, 1998. [Online]. Available: http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.13.4415

[21] Q. Limbourg and J. Vanderdonckt, "Comparing task models for user interface design," in The Handbook of Task Analysis for Human-Computer Interaction, D. Diaper and N. Stanton, Eds. Mahwah: Lawrence Erlbaum Associates, 2004.

[22] H. Trætteberg, Model-based user interface design. Information Systems Group, Department of Computer and Information Sciences, Faculty of Information Technology, Mathematics and Electrical Engineering, Norwegian University of Science and Technology, May 2002.

[23] F. Paternò, C. Mancini, and S. Meniconi, "ConcurTaskTrees: A diagrammatic notation for specifying task models," in Proceedings of the IFIP TC13 International Conference on Human-Computer Interaction, ser. INTERACT '97. London, UK, UK: Chapman & Hall, Ltd., 1997, pp. 362–369.

[24] S. E. Hudson, B. E. John, K. Knudsen, and M. D. Byrne, "A tool for creating predictive performance models from user interface demonstrations," in Proceedings of the 12th annual ACM symposium on User interface software and technology, ser. UIST '99. New York, NY, USA: ACM, 1999, pp. 93–102.

[25] B. E. John, K. Prevas, D. D. Salvucci, and K. Koedinger, “Predictive human performance modeling made easy,” in Proceedings of the SIGCHI conference on Human factors in computing systems, ser. CHI ’04. New York, NY, USA: ACM, 2004, pp. 455–462.

[26] A. Cypher, “Eager: programming repetitive tasks by example,” in Proceedings of the SIGCHI Conference on Human Factors in Computing Systems, ser. CHI ’91. New York, NY, USA: ACM, 1991, pp. 33–39.

[27] A. D’Ulizia, F. Ferri, and P. Grifoni, “A survey of grammatical inference methods for natural language learning,” Artif. Intell. Rev., vol. 36, no. 1, Jun. 2011, pp. 1–27.

[28] Human Interfaces in Information Systems (HIIS) Laboratory. ConcurTaskTrees Environment. [Online]. Available: http://giove.cnuce.cnr.it/ctte.html [retrieved: 11, 2014] (2014)