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 a method 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. We validate our method in three case studies.**

Keywords-task; tree; generation; usage-based; traces.

I. INTRODUCTION

Task trees are a well-known method to model user interactions as, e.g., done in [1]. They provide a structure to define how interactions are intended by the interaction designer [2]. They can also be used for comparing expected and effective user behavior as a basis for a semi-automatic usability evaluation [1]. Task trees are usually defined manually at design time [3]. For websites, they can also be generated based on existing Hyper-Text Markup Language (HTML) source code [4]. In both approaches, they 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. Such generated task trees represent the effective behavior of users and can, therefore, be used for usage analysis, e.g., in the context of usability evaluations. The results of a usage analysis can be 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 all kinds of desktop applications.

Task models are used to describe user actions. 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) [5], TaskMODL [6], and ConcurTaskTrees [7] [8]. We reuse the basic concept of task trees, but apply it in a simplified manner.

There have been several attempts to generate task trees automatically. For example, the Convenient, Rapid, Interactive Tool for Integrating Quick Usability Evaluations (CRITIQUE) [9] creates GOMS models based on recorded traces. A similar approach is proposed by John et al. [10]. ReverseAllUIs [4] generates task trees based on models of the Graphical User Interface (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 [11]. 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 [12]. This is not feasible for our approach as the recorded user actions do not follow such a structure. For example, a user may interrupt a task execution, which would lead to an incomplete sentence.

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 and present three case studies in which we tested the feasibility of our approach in Section IV. We conclude with a discussion and an outlook on planned future work.

II. TRACE-BASED TASK TREE GENERATION

In this section, we introduce our process for generating task trees. We commence with the definition of terms that we use in this paper. Then, we describe our approach of tracing users of a website. Finally, we provide details about the generation of task trees based on the traces.

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. For example, clicking with a mouse causes an `onclick` event. Typing a text into a text field causes an `onchange` event on the text field. Events are a representation 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, a user has to perform several actions. For example, for logging in on a website, a user must type in a user name and a password into two separate text fields and click on a confirmation button.

 ISBN: 978-1-61208-325-4

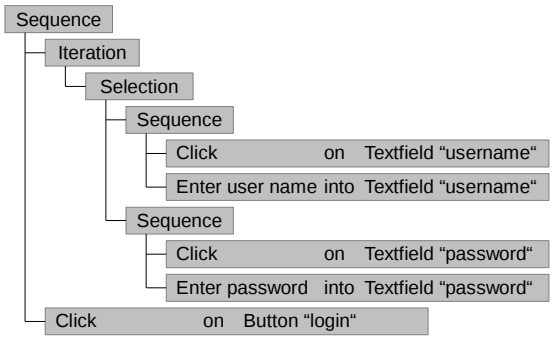

Figure 1. Example for a task tree

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 a user 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 structure the overall task into subtasks.

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 [8]. In our work, we consider the temporal relationships *sequence*, *iteration*, and *selection*. If a task is a sequence, its children are executed in a specified order. If a task is an iteration, it has only one child, which can be executed zero or more times. If a task is a selection, only one of its children is performed. 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 1. 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. The user may enter and change his user name and password several times. The overall task is completed after the user clicks the login button.

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 a user. We achieve this by integrating a monitoring module in the website. This module is invisible to the user and has minimal effect on the implementation, performance, and stability of the website [13]. 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.

A simplified example of a trace is shown in Figure 2. It lists the events recorded for a login of a 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, he reenters it a second time.

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 2. Example for a trace

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 3. Levels of design

C. Task Tree Generation

To describe the process for generating task trees based on traces, we introduce the levels of design, which are important for structuring task trees. We then describe the creation of the initial task tree, which is afterwards refined and condensed using temporal relationships.

1) Basic Approach: When designing GUIs, four levels of design are considered: conceptual design, semantic design, syntactical design, and lexical design. They are shown in Figure 3. The conceptual design describes the types of entities that are to be edited with a software [14], as well as their relationships [15]. 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.

The semantic design specifies functions to edit the entities defined in the conceptual design [14]. 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 [14]. 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 [14]. 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 a task for executing that function. Hence, there is 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 normal tasks with the constraint of not having children and not defining a temporal relationship.

Using this basic approach, we create task trees starting from the leaf nodes, i.e., from the event tasks. For each event in a trace, we generate an event task. All event tasks are stored in an ordered list in the order the respective events were recorded.

 ISBN: 978-1-61208-325-4

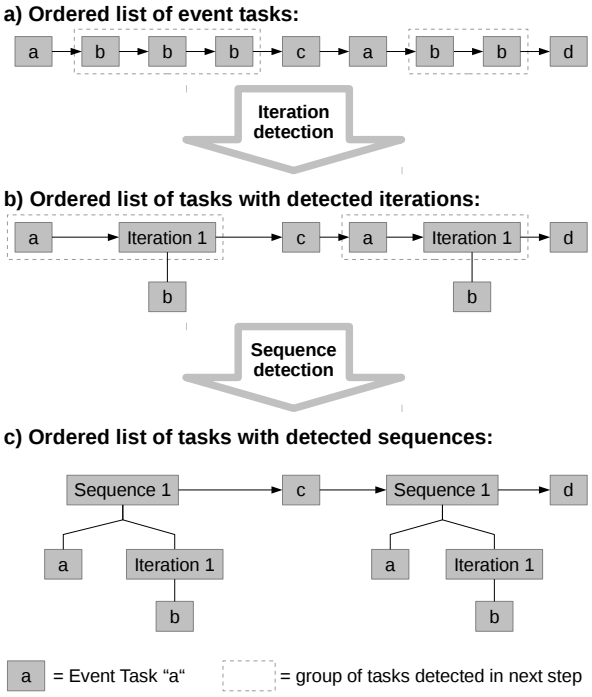

Figure 4. Example for the detection of iterations and sequences

An example is shown in Figure 4a where each grey rectangle denotes an event task and the arrows denote their order.

2) Iteration Detection: The ordered list of event tasks may contain identical tasks that occur subsequently. 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 tasks for iterations of identical tasks. If we observe an iteration, we generate a new task node of type iteration. This node gets the iterated event task as its single child. We then replace each occurrence of an iteration of the event task in the ordered list with the new iteration task node. Several subsequently occurring identical event tasks are herewith replaced by a single task node of type iteration. An example for this approach is shown in Figure 4. There, Event Task *b* is iterated several times (denoted by dotted boxes in Figure 4a). We replace these occurrences in the task list with single iteration nodes (Figure 4b).

3) Sequence Detection: After the iteration detection, we scan the list of tasks for identical 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 subsequence. Each occurrence of the identified subsequence in the task list is replaced with the new sequence task node. An example is shown in Figure 4. There, the subsequence of Event Task *a* and Iteration *1* occurs most often (two times) and is, therefore, replaced through task nodes representing this sequence.

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.

4) Repetition of Detections: The iteration and sequence detection on the list of tasks are repeated alternately until no more replacements are done. 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 4c the detected sequence contains a previously detected iteration.

If no more iterations or sequences are detected, the algorithm stops. The resulting task list contains detected task trees as well as event tasks, which were neither iterated nor part of a sequence occurring more than once. 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.

D. 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 [16]) 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. However, this work is still in its infancy and, therefore, not described in more detail.

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) [17]. 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 an 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. For this, it uses techniques from GUI testing. A popular approach for GUI testing is capture/replay [18], a technique where the tester interacts with the software and a capture tool records the executed actions. Afterwards, the actions are automatically executable with a replay tool in order to generate automated software tests. AutoQUEST uses the capturing to trace users and we reused these capabilities for our implementation for tracing users of websites.

 ISBN: 978-1-61208-325-4

```
<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 5. Example for a trace recorded with AutoQUEST's HTML monitor

AutoQUEST provides several platform specific plug-ins to trace the usage of software. This includes plug-ins for the Microsoft Foundation Classes, Java Foundation Classes, and websites. All plug-ins are comprised of a monitor to trace software usage and a trace parser to feed the recorded events into AutoQUEST. The monitors can be integrated with minimal effort into the software to be monitored. For example, 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 provided with AutoQUEST. After the integration of the JavaScript in the website, 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 which stores them into log files.

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 5. 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, we extended AutoQUEST with capabilities to generate task trees based on traces. The implementation follows the overall process described in Section II-C. The implementation of the iteration detection is straightforward and, therefore, not described in more detail.

1) Sequence Detection Implementation: For identifying and counting subsequences occurring several times, we reused and extended a data structure provided with AutoQUEST called trie [13]. 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 6.

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 6 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 6 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 6 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 6 represents the event tasks for the trace of Figure 2. 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 6, 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 6. 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.

IV. Case Studies

For the validation of our approach, we performed three case studies. For the first case study, we traced the interaction of users of our research website [19]. We integrated the HTML monitor of AutoQUEST in our content management system. We then recorded interactions of more than 700 users over a period of 6 months. Afterwards, we fed the gathered traces containing more than 25,000 events into AutoQUEST and generated over 600 task trees based on this. 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. From there, they navigated to the information about a specific lecture.

 ISBN: 978-1-61208-325-4

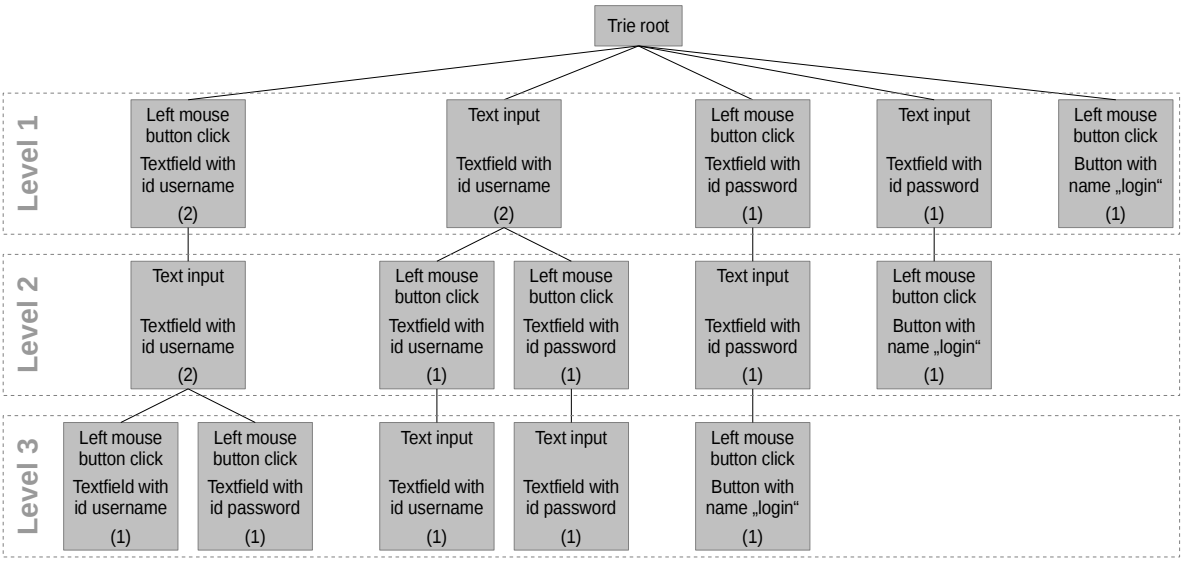

Figure 6. Trie generated based on the trace in Figure 2

The first case study also 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 3 months. This case study traced over 500 users producing more than 150,000 events resulting in 5,320 generated task trees. When feeding this data 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.

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. The other task trees showed the usage of the tabulator key instead. A visualization of the second login variant as displayed by AutoQUEST is shown in Figure 7. This presentation is a further extension done for AutoQUEST in the context of our work. 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 view and the users started the login process again.

In our third case study, we developed a sandbox example to validate the task tree generation with a subsequent usability evaluation. The sandbox contains several distinct views, each focused on a specific usability defect. We recorded our interactions on the sandbox, generated task trees, and performed an automated usability evaluation. The case study showed, that an automated usability evaluation is possible in general. However, the current implementation produces many false positives and is not mature enough to be described in more detail.

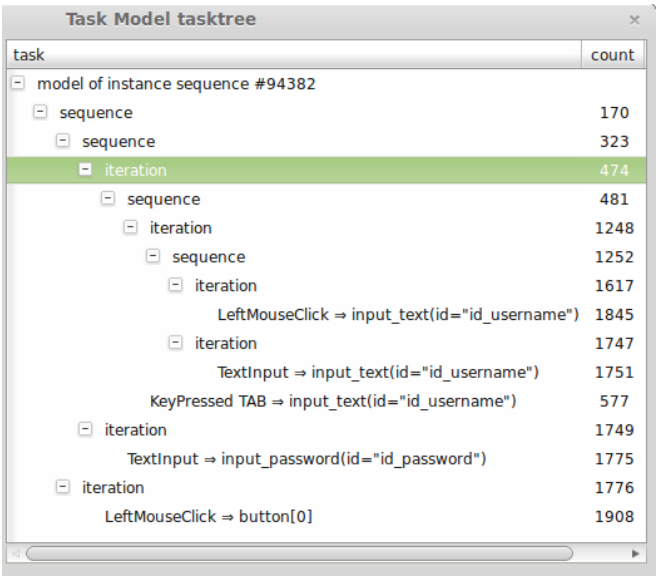

Figure 7. Task tree generated in the context of the second case study

V. Discussion of our Method

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. Currently, our approach is not able to identify distinct ways of executing semantically equal tasks. As an example, the different ways of filling out the login form in the case studies are treated as different tasks in the generated task tree.

 ISBN: 978-1-61208-325-4

At each repetition, the detection of subsequences chooses the longest sequence occurring most often and replaces it as described. This heuristic prefers shorter sequences as the count decreases with an increasing sequence 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.

VI. Summary and Outlook

In this paper, we described a method for generating task trees based on tracing user interactions. We implemented this method for websites and performed three case studies to validate its feasibility.

In our future work, we will improve and extend the task tree generation. We especially focus on the detection of an enhanced set of temporal relationships not considered in our work, yet. An example is the detection of selections of different approaches for executing the same task. We also plan to support a manual merging of such tasks to be able to treat them as identical in a subsequent usage analysis. Furthermore, we plan to implement both, a better heuristic for detecting more intuitive subsequences, as well as a flattening algorithm for reducing the complexity of the generated task trees. 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.

Acknowledgment

This work was done in the context of the project MIDAS (Model and Inference Driven - Automated testing of Services Architectures).

References

[1] 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.

[2] 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.

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

[4] 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.

[5] 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.

[6] 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.

[7] 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.

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

[9] 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.

[10] 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.

[11] 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.

[12] 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.

[13] 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.

[14] 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.

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

[16] U.S. Department of Health & Human Services. Usability.gov - improving the user experience - guidelines. [Online]. Available: http://guidelines.usability.gov/ [retrieved: 1, 2014] (2013)

[17] S. Herbold and P. Harms, “AutoQUEST - Automated Quality Engineering of Event-driven Software,” submitted to the Testing Tools Track of the International Conference on Software Testing (ICST) 2013, unpublished.

[18] 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.

[19] Software Engineering for Distributed Systems Group. Software Engineering for Distributed Systems. [Online]. Available: http://www.swe.informatik.uni-goettingen.de/ [retrieved: 1, 2014] (2014)

 ISBN: 978-1-61208-325-4