WevQuery: Testing Hypotheses about Web Interaction Patterns

AITOR APAOLAZA, School of Computer Science, University of Manchester
MARKEL VIGO, School of Computer Science, University of Manchester

Remotely stored user interaction logs, which give access to a wealth of data generated by large numbers of users, have been long used to understand if interactive systems meet the expectations of designers. Unfortunately, detailed insight into users' interaction behaviour still requires a high degree of expertise and domain specific knowledge. We present WevQuery, a scalable system to query user interaction logs in order to allow designers to test their hypotheses about users' behaviour. WevQuery supports this purpose using a graphical notation to define the interaction patterns designers are seeking. WevQuery is scalable as the queries can then be executed against large user interaction datasets by employing the MapReduce paradigm. This way WevQuery provides designers effortless access to harvest users' interaction patterns, removing the burden of low-level interaction data analysis. We present two scenarios to showcase the potential of WevQuery, from the design of the queries to their execution on real interaction data accounting for 5.7m events generated by 2 445 unique users.

CCS Concepts: • **Information systems** → **Web log analysis**; • **Human-centered computing** → *Empirical studies in HCI*; HCI design and evaluation methods;

Additional Key Words and Phrases: Hypothesis Testing, A/B Testing, User Interface Evaluation, Usability, Web

ACM Reference format:
Aitor Apaolaza and Markel Vigo. 2017. WevQuery: Testing Hypotheses about Web Interaction Patterns. *Proc. ACM Hum.-Comput. Interact.* 1, 1, Article 3 (June 2017), 17 pages.
https://doi.org/10.1145/3095806

1 INTRODUCTION

When developing interactive systems user-centered design paradigms are not always followed and developers assume the prospective users of the system are like themselves – 'you are not the user' is the statement that exemplifies this phenomenon [8]. This typically incurs in usability problems being introduced into interactive systems, which is detrimental for the effectiveness of the developed system and for the experience of the users. On systems that are already built and deployed summative usability tests are used to catch these flaws and take further action.

Over the years Human-Computer Interaction has established a plethora of methods to isolate and identify existing usability issues in interactive artefacts [16]. All of them have strengths and drawbacks: for instance, if we discuss the setting of the study (i.e. laboratory vs. remote) laboratory studies provide a deep understanding of

This work was supported by the EU's Horizon 2020 research and innovation programme under grant agreement H2020-693092 MOVING http://moving-project.eu and the EPSRC [EP/I028099/1].
The code for WevQuery at the point of this paper's submission is available at https://github.com/aapaolaza/WevQuery/commit/7186881
Author's addresses: A. Apaolaza and M. Vigo, School of Computer Science, Oxford Road, University of Manchester, M13 9PL, Manchester, United Kingdom.
Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
© 2017 Copyright held by the owner/author(s). Publication rights licensed to Association for Computing Machinery.
2573-0142/2017/6-ART3 $15.00
https://doi.org/10.1145/3095806

interactive systems and allow for controlling confounding variables. However they are resource demanding and they typically follow a task-oriented experimental design, which increases the internal validity of the outcomes but tend to be detrimental to the external validity. They are also restrictive in that they cause the 'Guinea Pig' effect [26] whereby users feel observed and do not behave as they would normally do.

Non-controlled remote interaction data can help tackle these hurdles, but introduces additional challenges. The context of data captured *in the wild* is often unknown or costly to retrieve as it cannot be controlled; this means, for instance, that we do not know the task being realised, whether users are being interrupted while carrying out their tasks or whether their information needs changed over the course of the interaction. The lack of information about the users' specific setup, their goals and motivations, makes obtaining any insight about their interaction even more challenging. Nevertheless, the analysis of non-controlled remote data provides further advantages, such as highly ecological interaction data, more affordable experimental setups and access to a larger target population. In line with this, the analysis of user interface events has traditionally been used as a way to understand how individuals use interactive systems [13] and is one of the main methods employed in remote studies.

Popular tools such as Google Analytics provide a high-level view of remotely captured interaction, making the analysis accessible. Although useful, such approaches are mainly based on clickstreams that overlook the nuances of users' interaction. While coarse interaction data allows designers to identify tasks with unusually long execution times, low-level interaction data provides detailed insights to understand the reason for these delays. For instance, designers can identify points in which the users systematically hover elements over prolonged periods of time, scroll unnecessarily, or remain inactive for a prolonged period of time (e.g. user is *away from keyboard*). Unfortunately, the analysis of such low-level logs requires a high degree of expertise and domain specific knowledge.

We make the analysis of low-level interaction logs accessible by providing a graphical tool to design queries interactively. WevQuery[1], which stands for Web Event Query Tool, allows designers, who may not necessarily be database experts and have programming skills, to build queries to retrieve information about behaviours exhibited on the Web. These queries are represented as sequences of events that are defined using interactive drag-and-drop functionalities on a Web application. The queries can include a number of user interface events including scroll change events and mouse interaction. Temporal relations between the events can also be incorporated into the query: e.g. one can define the minimum time elapsed between two consecutive events etc. The system then translates this graphical representation into a query and searches for patterns of events that match the defined sequence.

Consequently, WevQuery is conceived as a tool for hypothesis testing on the Web. After formulating hypotheses about user behaviour on a Web system, designers can test the hypotheses and retrieve detailed information about the requested behaviour. This not only helps in making design decisions based on empirical evidence, but it also informs the specification and formulation of hypotheses for A/B testing experiments. As a proof of concept we discuss the strengths of WevQuery in two scenarios where two designers discuss the possible effects certain aspects of the interface might have on users' interaction with a large Web system. These scenarios illustrate the advantages of WevQuery from the design and execution of the queries, to the interpretation of the results and the possibilities for iterative refinement of the hypotheses.

The contributions of the paper are:

- We present WevQuery, a platform that allows designers to test and refine their hypotheses about how users interact on Web systems.
- We show the strengths of WevQuery in terms of flexibility, scalability and performance on a real website where 2 445 recurring users generate 5.7m events on 3 287 Web pages for a period of two months.

[1] Available at https://github.com/aapaolaza/WevQuery

- We discuss the usefulness of WevQuery on real scenarios where two designers with different points of view use WevQuery to empirically support their arguments.

2 RELATED WORK

Laboratory studies are a common approach to testing the usability user interfaces. They have several advantages including the direct interaction with participants, opening up the possibility of obtaining feedback during the observation. The use of bespoke observation methods, not available in the users' environment, is another advantage of the laboratory setting. Laboratory studies have also a number of drawbacks: they can be obtrusive as they alter the users' environment. Displacing users can alienate them and modify the way they would usually behave. The 'Guinea-Pig' effect and the response bias [26] can cause further problems too. Remote tests are known to be as effective as laboratory tests if they are controlled and run synchronously between the experimenter and the participants [3].

Naturalistic remote approaches address some of these issues and provide access to a wider pool of participants at the cost of a less controlled environment, where confounding variables cannot be isolated. Web logs provide the most cost-effective approach to obtaining remote data, as they typically do not require any substantial change to the system. On the other hand, significant effort is needed to preprocess and obtain relevant information from these logs [27]. When the sample size of users is large, and particularly when testing complex interactive interfaces, manual analysis can be challenging. Automated approaches can help if a suitable level of abstraction on user activities is provided [13]. Whereas detailed visualisations of users' interaction have been found to help designers understand user interaction [7, 20], their main problem is their lack of scalability and capacity to work with large amounts of data. Also there should be some expectations about the tasks users carry out and sometimes, especially in naturalistic remote studies, this is unknown.

Commercial approaches such as Google Analytics[2], or the open source alternative Piwik[3], provide insight into referrals and demographics, collecting this information through the inclusion of JavaScript code on the pages that are tracked. The strength and popularity of this approach is corroborated by numerous studies [6, 19] whereby interaction data can be captured over prolonged periods of time to later categorise large pools of users according to their behaviour [22, 24]. The aggregation of user interface events from large pools of users has also been used to enable the systematic evaluation of user interfaces [18]. Aggregating user interface events across users makes the analysis of data scalable in some of the above-mentioned approaches. However, this often entails that the behaviours of particular users may get diluted when analysing data in this fashion, which suggests there is a trade-off between the efficiency of the analysis and the granularity of the behaviours designers are interested in.

Some have suggested that particular sequences of use or activities are indicators of errors [25]. For example, undo and erase actions in 3D design tools have been successfully employed as a way to detect interaction errors that would otherwise remain unreported [1]. The isolation of errors helps to recreate them so that they can be repaired. Other sequences can be less intuitive indicators of problems: for example, repeated actions on the interface can signal users failing to perform an action [17].

2.1 Querying Event Data

Complex event processing [10] systems detect instances of particular patterns of events in event data streams, so particular actions can be triggered accordingly. For example, an increase in stock values can trigger a purchase action, and a user being idle for too long can trigger a help dialogue. Despite the potential of using event data, there is a lack of support to design effective queries, especially for those who are not specialists. Existing languages for event querying [9] allow designers to establish relations between events and define temporal constraints. Some

[2]https://analytics.google.com
[3]https://piwik.org/

domains are more prone to using these technologies: for instance, medical information systems generate a large number of events, which are, typically, semantically meaningful. Examples of these events include prescriptions, medical conditions, and visits to the hospital. In this regard, visual query systems have been proposed to query such events [12], where temporal patterns can be defined specifying the order and time intervals between events. Other works have taken an existing well established syntax, such as SPARQL, and have extended it with temporal and order operators [4]. The analysis of logs containing user interface events can be understood as a particular case of complex event processing so many of the techniques may be transferable.

Although the advantages of using user interface event data are well known, the lack of support to create event based queries prevents designers from performing an iterative exploration of their datasets. Some approaches have tackled this lack of support by designing query languages that incorporate aspects particular to user interface event data. This simplifies the query design process, relaxing the demands of the required skills [14]. Unfortunately, simplification entails decreasing the expressiveness of the queries, leaving out key features such as temporal relations between events. The (s|qu)eries tool [28] makes use of the capabilities of regular expressions, including alternate paths – via logical disjunctions – and event repetitions. This approach scales with difficulty as it requires storing the user interface event dataset in memory. While it also reports temporal metrics on the results, such as duration of the pattern, establishing temporal constraints in the query is not possible.

In a nutshell, languages for querying events provide features to create queries by including temporal constraints although using these languages requires further learning. Existing interactive query systems have been found to be too complex, and not easily applicable to Web interaction. Some of these systems sacrifice flexibility for ease of use, preventing the design of complex queries. Other highly interactive approaches support the iterative exploration of event patterns, but scalability has not been taken as a factor. Consequently, the identified shortcomings suggest that these limitations need to be addressed in order to provide support for a user interface event querying system which is straightforward, flexible, extensible and scalable. WevQuery tackles these limitations through the provision of:

- A visual framework that supports designers in querying user interface event data without having to learn another language.
- An extensible and scalable query-processing engine that makes use of the MapReduce paradigm.
- Human and machine-readable reports for decision making.

3 ARCHITECTURE OF WevQuery

Figure 1 shows the general architecture of WevQuery, which consists of two main components: first, an interface with an interactive Web application that allows designers to graphically define their queries or hypotheses; second, the back-end translates these graphically designed hypotheses into queries that are run against a database and produces a comprehensive report about the formulated hypotheses.

Figure 2 is a screen capture of WevQuery's graphical interface and shows the functionalities designers can use to construct the queries that test their hypotheses. Queries are defined as a sequence of events and, when executed, look for patterns of events matching that sequence. Temporal restrictions can be included in order to establish intervals of time between events that match the query. The graphical interface is dynamically built from an XML Schema that defines the grammar of the queries, which ensures the resulting query complies with the requirements of the system. Such queries are stored as XML files that conform to the mentioned schema so that designers can reuse and share them at any time. Then the query in the XML file is transformed into a MapReduce query. MapReduce [11] is a programming paradigm designed to handle large datasets. MapReduce makes use of two functions: the *map* function splits the data into subsets and then the *reduce* function processes each subset independently. We have chosen this paradigm to ease the processing of large amounts of interaction data and make it scalable and suitable for distributed systems. This paradigm is also particularly suited for WevQuery, as

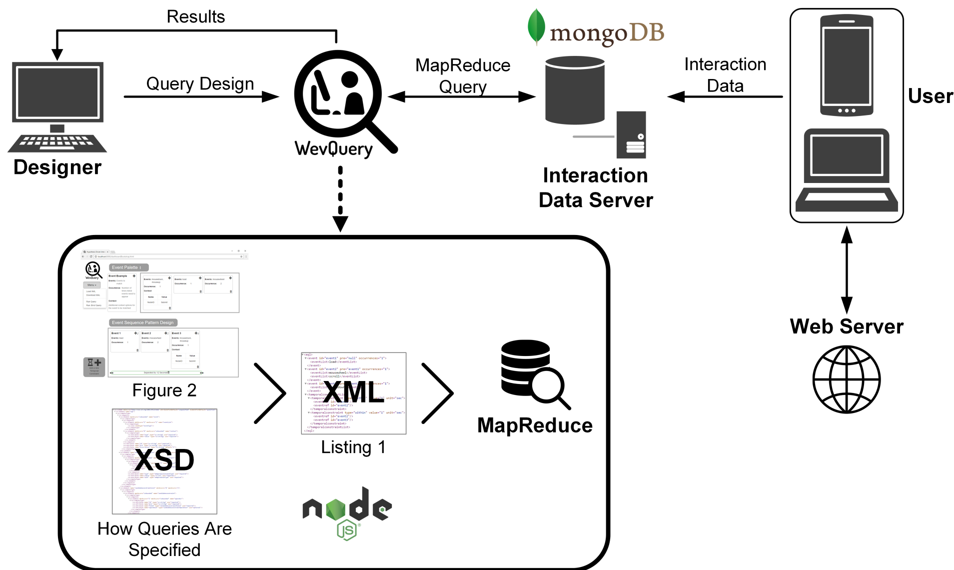


Fig. 1. Architecture of WevQuery

interaction data from individual user episodes is processed independently. The MapReduce query is then run against the Interaction Data Server, which stores user interface events in a MongoDB[4] database. WevQuery runs on Node.js[5], which is used to host the Web application serving the graphical interface as well as to execute the designed queries and connect to the MongoDB database.

3.1 Capturing User Interface Events

User interface events are captured using a modified version of UsaProxy [6] that improves the deployability and scalability of the original system [5] – the capture tool is publicly available[6]. The data capture module also uses MongoDB to store the captured user interface events. Captured events include all mouse and keyboard interaction, as well as browser window events, changes to the state of the DOM (Document Object Model) elements on the page and other system information including screen resolution. The system has been tested using interaction data captured from a publicly available website[7] with thousands of accesses per month over a period of two months. A total of 2 445 unique users have been identified, who have generated a total of 5 702 868 low-level events.

3.2 Defining the Queries

Queries are formally specified through a set of rules that define the values and attributes of user interface events and the relations between different events. Relying on a formal structure makes possible to automatically transform the queries into scalable database queries. Such formalisms were defined using an XML Schema that was inspired by works that identify requirements of complex models in event-based systems [21]. The use of XML provides multiple advantages: it can be easily validated against the defined structure, it is human readable,

[4]https://www.mongodb.com/
[5]https://nodejs.org/
[6]https://github.com/aapaolaza/UCIVIT-WebIntCap
[7]http://www.cs.manchester.ac.uk/

it can be easily extended with additional query features, and its use is suitable for programmatically deriving MapReduce queries. As above-mentioned the graphical interface shown in Figure 2 also uses the XML Schema to generate the graphical elements that allow to define interaction patterns on-the-fly based on the current definition of the query.

Taking into account the sequential nature of the captured interaction events, our current schema implements a subset of the possible relations between user interface events [2]: *precedes* and *preceded by*. These relations indicate either the following or the preceding event within a set of events arranged in a particular order and discarding overlaps.

Interaction events are captured synchronously in a strictly ordered manner. Overlaps between these interaction events are not possible as interaction events are pinpointed in time and they are atomic. An example of a query designed with WevQuery can be seen in Listing 1. The order between the events is set by declaring each events' predecessor using the `pre` attribute while remaining attributes store values of the query. The query in Listing 1 is designed to test a hypothesis formulated in Scenario 2 in Section 4. The various event elements describe the pattern to search: a single `load` event, followed by a single occurrence of either `mousewheel` or `scroll`, ending on a single `mousedown`. `temporalconstraintList` element contains two `temporalconstraint` that restrict the query by defining temporal relationships between events: these constraints ensure that the time elapsed from `event1` to `event2` and from `event2` to `event3` is less than 1 second. This is conveyed by the `within` value of the `type` attribute, which sets the scope of the interval, and specified by the `value` and `unit` attributes.

3.3 Defining Queries through WevQuery's Web Interface

The graphical user interface shown in Figure 2 enables designers to build queries. This interactive Web application supports the design of the queries, allowing designers to drag and drop the event elements, and automatically showing the available values for the attributes of the events. The queries are composed of a sequence of ordered events and each event in the sequence can match one or more types of interaction events. For example, a particular event in the sequence can be set to match either a `mousedown` (describes the action of clicking the mouse) or a `mousewheel` (describes the interaction with the scroll wheel of the mouse). In the case of the first event from the Event Palette in Figure 2, it matches two different event types: `mousedown` and `mouseup`. The graphical interface consists of several modules including an event palette, a widget to define sequences and components to establish the temporal relationships between events.

3.3.1 The Event Palette. This widget displays the user events that can be selected when defining a query. When the designer presses on the *plus* ✚ sign in the "Event Example" box (under the Event Palette header in Figure 2), the event template creation dialogue is shown as depicted in Figure 3a. One or many event types can be selected, which are loaded from the XML Schema described in Section 3.2. The number of times an event type needs to be matched can be set in the *occurrence* field.

For example, a `mousedown` event with an occurrence value of 2, is equivalent to a sequence of two independent `mousedown` events with occurrence value of 1. The context for the match can also be set by specifying the element of the user interface that triggered such event, using the attributes of the DOM node to do so. It can also set the scope of the Web page where the event took place by specifying its URL(s). Once the event is defined it is added to the palette so that it can then be dragged into the widget that allows the specification of event sequences – see below.

3.3.2 Designing Patterns of Event Sequences. Events created in the Event Palette can be dragged and dropped into the Event Sequence Pattern Design area (bottom panel in Figure 2), using the *move* icon at the top-right of the element that represents an event. The position of events in the list determines the order of the sequence which is conveyed by a number located next to the mentioned *move* icon. The resulting query consists of a

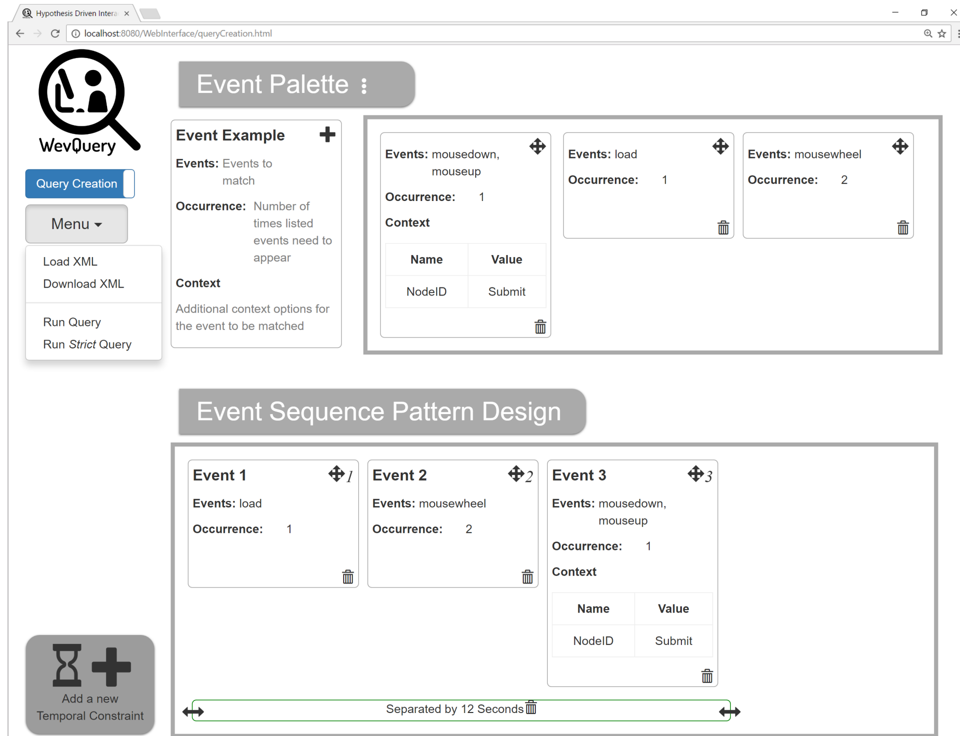


Fig. 2. Screenshot of the WevQuery Web application

sequence of events WevQuery uses to look for patterns that match the sequence. Events can also be discarded by clicking on the *bin* icon located at the bottom-right corner of each event.

3.3.3 Defining Temporal Constraints. The addition of temporal constraints allows designers to set time intervals between matched events. If not specified, the query will ignore the time elapsed between events. When clicking on the "Add a new Temporal Constraint" button, the dialogue that shows in Figure 3b pops up, allowing designers to establish the following temporal aspects:

- **Relation** determines if the temporal distance between selected events has to be under (within) or above (separated by) the indicated threshold.
- **Events** allows the designer to select the two events affected by the temporal constraint. When one of the buttons is pressed, the dialogue temporarily disappears so that the user can select the events in the Event Sequence Pattern Design area (bottom panel in Figure 2).
- **Duration** and **Unit** determine the temporal distance, and the unit of time that designers wish to establish.

Once temporal constraints are defined, the length of the bar that conveys the scope of temporal constraints can be dragged and modified.

3.3.4 File Menu. The file menu at the top left corner of the WevQuery interface (see Figure 2) allows designers to operate with the designed query: once the query is defined, the designer can *Run Query* and visualise the results on the screen, or receive them via email (a prompt is shown to provide the email address).

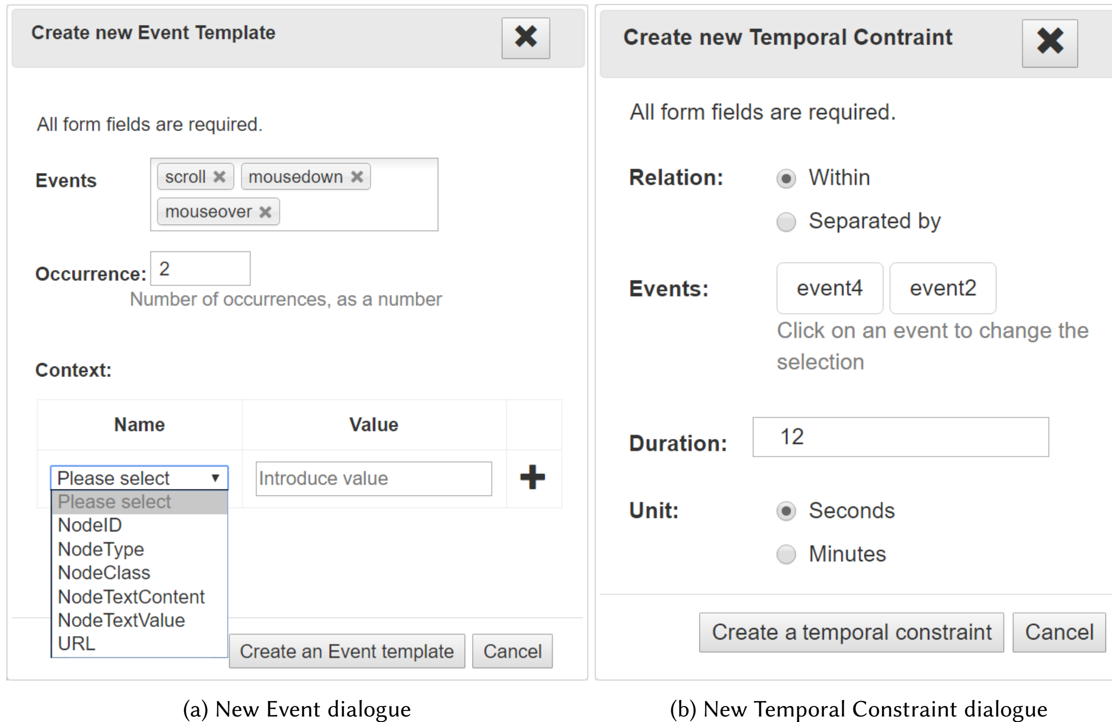
```xml
<eql>
  <event id="event1" pre="null"
      occurrences="1">
    <eventList>load</eventList>
  </event>
  <event id="event2" pre="event1"
      occurrences="1">
    <eventList>mousewheel</eventList>
    <eventList>scroll</eventList>
  </event>
    <event id="event3" pre="event2"
        occurrences="1">
    <eventList>mousedown</eventList>
  </event>
  <temporalconstraintList>
    <temporalconstraint type="within"
        value="1" unit="sec">
      <eventref id="event1"/>
      <eventref id="event2"/>
    </temporalconstraint>
    <temporalconstraint type="within"
        value="1" unit="sec">
      <eventref id="event2"/>
      <eventref id="event3"/>
    </temporalconstraint>
  </temporalconstraintList>
</eql>
```

Listing 1. Example of query created with WevQuery

```json
"_id" : {
  "userID" : "2HZhjN5yQjAC",
  "url" : "http://www.cs.manchester.ac.uk/",
  "episodeCounter" : 1
 },
"value" : {
  "xmlQuery" : [
   [{
     "event" : "load",
     "timestamp" : "2016-04-13,17:13:53",
     "timestampms" : 1460564033927,
     "htmlSize" : "960x1054",
     "resolution" : "960x600",
     "size" : "960x431",
    },
    {
     "event" : "scroll",
     "timestamp" : "2016-04-13,17:13:54",
     "timestampms" : 1460564034166,
    },
    {
     "event" : "mousedown",
     "timestamp" : "2016-04-13,17:13:54",
     "timestampms" : 1460564034977,
     "mouseCoordinates" : {
       "coordX" : 361,
       "coordY" : 294,
       "offsetX" : 22,
       "offsetY" : 8
     },
     "nodeInfo" : {
       "nodeDom" :
           "id(\"Main\")/DIV[1]/A[2]",
       "nodeLink" : "/research/",
       "nodeText" : "Research",
       "nodeType" : "A",
       "nodeTextContent" : "Research",
       "nodeTextValue" : "undefined"
     }
    }]
  ],
  "isQueryStrict" : false
 }
```

Listing 2. JSON report generated as a result of a query

A variant of the standard query, *Run Strict Query* ensures that no non-matching events are found between the events in the sequence. For example, a query can look for a mouse click, a mousewheel event and a keypress, in that order. If run strictly, any sequence of events where a mouse click is found between the mousewheel and the keypress will be discarded. Is important to take into consideration that when the query is translated into a MapReduce query, only the events selected in the sequence are retrieved from the database in order to find possible matches for the query. This filtering significantly reduces the execution time of the query. Once the



(a) New Event dialogue

(b) New Temporal Constraint dialogue

Fig. 3. Additional dialogues of the WevQuery interface

query is defined it can also be downloaded as an XML file so it can be reused and shared. An example of such file describing a query can be seen in Listing 1.

3.4 From XML to Querying the Database

To run the resulting query the XML file is first transformed into a MapReduce query that is executed against the Interaction Data Server (see Figure 1). The sequences of events and temporal conditions are converted into JavaScript objects so that the MapReduce algorithm can handle them. In other words, the query sequence is transformed into an array of event objects. For each event, the event type and its context (as defined in the event creation dialogue in Figure 3a) is stored, as many times as its *occurrence* indicates. For example, the sequence described in the Event Sequence Pattern Design area of Figure 2 would be transformed into: [`load`, `mousewheel`, `mousewheel`, `mousedown/mouseup`]. The resulting query looks for four matching events, as the value of the occurrence of *Event 2* was 2.

The temporal constraints are also transformed, and the indexes of the events affected by the constraint are set to the first occurrence of the corresponding event. The query then searches for the specified pattern of events throughout all users' episodes. WevQuery defines episodes as the interaction from a single user that happen in a Web page without a *noticeable* interruption, where any interruption longer than 40 minutes is considered noticeable. This threshold is consistent with what the literature suggests as far as the definition of episodes is concerned [15, 23]. MapReduce is suited for this task as it allows to split the execution of the algorithm for each episode. This way, the algorithm can analyse the interaction events contained in each episode independently.

ALGORITHM 1: Simplified matching algorithm to process the events of each episode

```
1  for currentEvent ← Interaction Events in Episode do
2      for currentCandidate ← List of Candidates do
3          if currentEvent == currentCandidate.nextEvent then
4              Add currentEvent to currentCandidate.eventList;
5              if temporal constraints are correct then
6                  if currentCandidate matched all events then
7                      Add currentCandidate to results;
8                      Remove currentCandidate;
9                  end
10             else
11                 Remove currentCandidate;
12             end
13         else if execution is Strict then
14             Remove currentCandidate;
15     end
16     if current event matches first event in sequence then
17         Create new candidate;
18         Add candidate to List of Candidates;
19     end
20 end
```

The algorithm processes every interaction event in the episode in chronological order – Algorithm 1 shows the pseudocode that describes such process. Every time the first event in the sequence is matched, a new candidate pattern is created. This candidate pattern contains a queue with the full list of events to be matched. Every time an event is processed (*currentEvent* in line 1), all created candidates check if their next event in the queue can be matched (*currentCandidate.nextEvent* in line 3). For an event in the queue to be correctly matched, the processed interaction event must be in the list of accepted event types. If the event contains any context, then this context needs to be matched as well. For example, the list of accepted event types for the Event 3 in the Event Sequence Pattern Design Area in Figure 2 are `mousedown` and `mouseup`, and the `NodeID` field needs to have the value `Submit`. Once all events in a candidate pattern's queue are matched, and as long as the candidate pattern complies with all the temporal constraints, the candidate pattern is stored as a result. If the temporal constraints are not satisfied, then the candidate pattern is discarded. Finally, if the query is *Strict* a mismatch will cause the immediate rejection of the candidate pattern (line 13).

Matching Algorithm Example. In this example the designer creates a query for the following sequence: `mousewheel` with occurrence value of 2 followed by a `mousedown`/`mouseup` on a `Submit` button. The query is transformed for the matching algorithm as shown in Figure 4. In the same figure, three different inputs are described, as examples of possible interaction sequences found in an episode. Input 1 represents a successful match, and is accepted as a result at the third step. In Input 2 a successful match is found at the fourth step if the query is not run in strict mode. If the query is strict, the corresponding candidate is rejected at the second step (a `mousewheel` could not be matched). In Input 3, if the query is strict, the corresponding candidate is rejected at the second step. If the query is not strict, the corresponding candidate is rejected at the fourth step as the node value is different to that of the query.

Fig. 4. How the query matches different inputs through Algorithm 1

The results of the query, which consist of a JSON and a CSV file with all the matching sequences of events, can be visualised on the screen or sent to the designer via email. Each report contains detailed information about the circumstances in which sequences of events were triggered. The number of episodes in which the pattern is found is reported as well as a list of the details of each matched event. Listing 2 illustrates a report containing one occurrence of a pattern that matched the query. In this simplified report, the context of the occurrence (the user, the URL, and the episode where the occurrence took place) and details of the matched events can be retrieved. Details include temporal information, as well as event specific information. For example, in the case of the page `load` event, the page size is retrieved, and in the case of the `mousedown` event, details about the mouse coordinates, and the target element the user interacted with are provided. In the case of these results, the designer can see that it took the user 1 second to click on the element "Research" after loading the page.

4 SCENARIOS

To illustrate the potential of WevQuery, we present two scenarios in which two designers discuss changes they have made to a Web system, and how these may affect user interaction. The investigated website is the publicly accessible website of the School of Computer Science at the University of Manchester[7] and real interaction data has been used to illustrate the scenarios below.

4.1 Scenario 1: Investigating the Effect of Interactive Tiles

The website under investigation contains some Web pages with a tiled interface. These tiles change when a user hovers them: the common behaviour is to reveal a different image, zoom into the current one, or change the style of the tile as a visual feedback of it being hovered. Designer A decided it would be a good idea to instead disclose some links and additional content (see Figure 6). Designer B disagrees with this addition, arguing that users will spend a lot of time needlessly hovering elements for prolonged periods of time to determine if there is some hidden content.

In order to determine who is right, they design a query that will search for all periods of time in which any element was hovered longer than 3 seconds as specified by WevQuery in Figure 5. The query is executed and from the 29 770 episodes that match with the query (`mouseover` and `mouseout`), they learn that in half of the sequences (i.e. 14 850 cases) users actually hovered longer than 3 seconds. As they realise that 3 seconds might be a too low threshold, and could be returning non-problematic behaviours, they increase the specified interval to

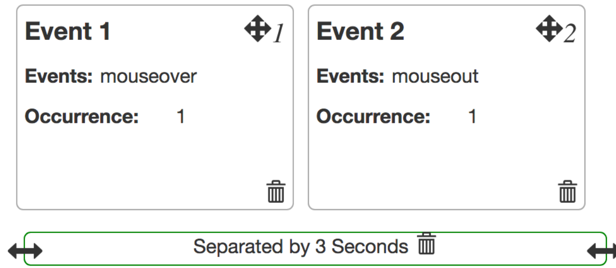
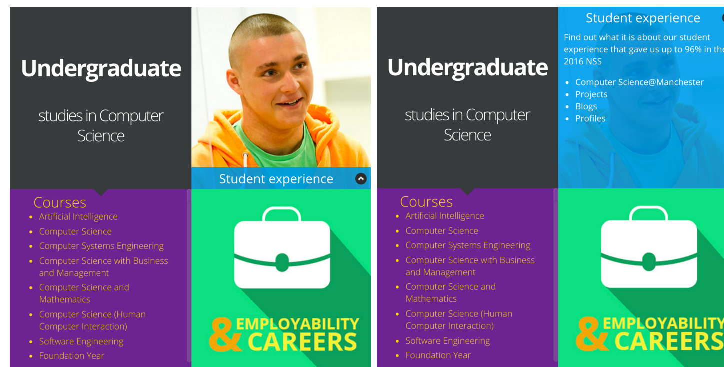


Fig. 5. Example of the Scenario 1 query designed with the WevQuery interface

```
<eql>
  <event id="event1" pre="null"
      occurrences="1">
    <eventList>mouseover</eventList>
  </event>
  <event id="event2" pre="event1"
      occurrences="1">
    <eventList>mouseout</eventList>
  </event>
  <temporalconstraintList>
    <temporalconstraint
        type="separated" value="3"
        unit="sec">
      <eventref id="event1"/>
      <eventref id="event2"/>
    </temporalconstraint>
  </temporalconstraintList>
</eql>
```

Listing 3. Resulting XML for the Scenario 1 query

10 seconds and run the query again. Now, they find that 8 277 episodes exhibited mouse hover behaviours longer than 10 seconds. When looking closer to the report, they find out that this behaviour was exhibited on a particular Web page (i.e. sorting the URLs by number of occurrences, the homepage exhibited 539 occurrences, 5 times more than the second page according to that ranking) and the mouse mainly hovered <a> elements, commonly used for hyperlinks (i.e. 183 out of the 539 identified occurrences). When analysing the home page carefully they find out that these elements are being used to disclose more options to the users. They see that hovering one of the <a> elements can disclose as many as 45 different links. This amount of available options forced users to hover over these <a> elements for prolonged periods of time. Consequently they decide to use a toggled approach so that users click on the item to open the menu with further options. In this way users do not have to hover over those elements for so long. They are still not sure if this change will facilitate the intended behaviour from users so they decide to run an A/B test. They can then use WevQuery to compare the interaction behaviour data obtained from both variants.



(a) The tiles before hovering (b) The tiles after hovering

Fig. 6. Web elements tested in Scenario 1

4.2 Scenario 2: Investigating whether Users Check the Bottom Menu

Some pages on the website seem to be too large (or not responsive) for small monitors, hiding a bottom menu of links. Designer A believes this is not a problem as in this website most of the information at the bottom of the pages is not as relevant as the remaining content. Designer B argues that most of the users end up scrolling down right after the page is loaded in order to be able to have an overview of the page and see all the hidden information, which renders tall designs of the pages not effective.

In order to determine who is right they query for all the scrolling events that take place 10 seconds after the Web page load event. They query for both `mousewheel` events, that correspond to the use of the scroll wheel in the mouse, and `scroll` change events, which is triggered when the scroll state of the page changes. Adding the scroll change event as an alternative to `mousewheel` allow designers to include other interaction events that could affect the scroll state of the page, such as links within a page, or the use of the vertical scroll bar.

The results of the query indicate that from a total of 33 444 episodes where at least a page `load`, a `mousewheel` or a `scroll` event are found, 18 729 contain the queried pattern. They find this value to be quite high, so they refine the search using 0.5, 1 and 3 seconds as values for the temporal distance between the `load` and the `mousewheel/scroll` event. The results are 8 903, 11 885 and 16 129 episodes respectively.

Surprised with the high number of occurrences of this behaviour, Designer B hypothesises that the reason why this behaviour is so common is because users go to the bottom directly to click on some link they are familiar with and provides quick access to the resource they are looking for. Consequently they refine the query even more by adding a new event: they add a mouse click which takes place after the user scrolls – this would suggest that users scroll down to click on a specific link. By doing so they find 6 596 occurrences of this behaviour, approximately a third of all episodes. Then, they decide to add an extra temporal constraint that matches those occurrences of mouse clicks that take place within 1 second of the scroll action. The resulting query and its corresponding XML file can be seen in Figure 7 and in Listing 1 respectively.

On this occasion they only find 371 occurrences of this behaviour. Even if it is a relatively small number of instances, they decide to investigate the results to obtain more insights into why the users are exhibiting this behaviour. They discover that even if users scroll immediately after the page is loaded, the links they click afterwards are rarely at the bottom of the page but show up earlier. As a result they decide that both designers are correct, as the information at the bottom is not sought but having pages that do not fit the screen cause unnecessary interaction.

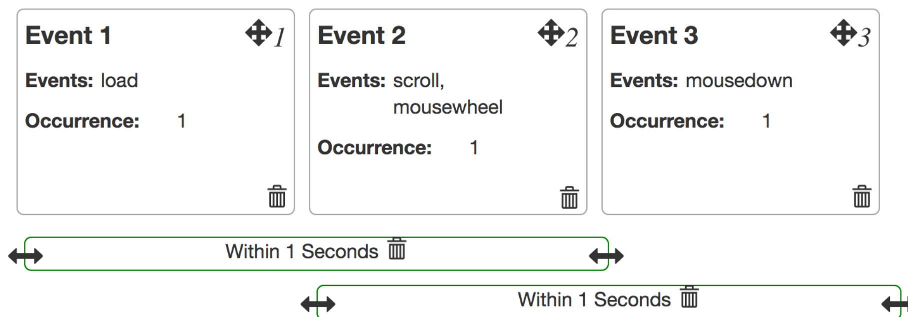


Fig. 7. Example of the Scenario 2 query designed with the WevQuery interface

4.3 On the Performance of Queries

The queries of the above scenarios were run on real interaction data as described in Section 3.1. The queries have been executed on the same testing environment where the Interaction Data Server is hosted. This testing environment runs on a Windows 10 64-bit operating system on an Intel Core i7, 16GB RAM and a 512GB solid-state drive storage device.

The execution time of the queries depends largely on the number of occurrences of the events that define the query. The hypothesis in Scenario 1 included the use of the `mouseover` and `mouseout` events. These events are specially frequent and the frequency is strongly dependent on the density of interface elements, as these events are triggered every time the mouse hovers an element of the interface. In the case that concerns us and its corresponding dataset, the combination of these two events accounts for more than half of all events: 2 791 488 from a total of 5 702 868 events in the interaction database, i.e. 49% of the events. The queries on this scenario consistently take 2 minutes to compute and the modification of the time constraint values has no impact on their performance.

In the case of Scenario 2, the selected events are less frequent. The `scroll` events (528 699 instances) and `mousewheel` (390 290 instances) are more frequent than `mousedown` (55 142 instances) and `load` (41 758 instances). The execution of the queries in this scenario takes 50 seconds. In this case, the impact of including the `mousedown` event into the query as well as the addition of temporal constraints is negligible.

4.4 Discussion

We have presented two scenarios describing the usefulness of WevQuery. With little knowledge of query languages, designers were able to test hypotheses about users' interaction and iteratively refine their queries. These examples highlight three main advantages of WevQuery:

- **Flexibility**. Previous approaches [4, 9] propose the use of query languages with event specific parameters, such as order and temporal relations. WevQuery removes the complexity of coding such parameters by providing an interactive user interface which is flexible enough to replicate the features of manual coding.
- **Extensibility**. In the scenarios we discuss, designers made use of interaction event types and complex temporal relations. The use of an extensible syntax to define the queries makes it possible to easily extend these functionalities. If we, for instance, wanted to define events from new data sources (e.g."Click on checkout button"), this would only require modifying the underlying XML Schema, which would automatically update the WevQuery interface and the MapReduce logic.
- **Scalability**. In order to support iterative testing there is a demand for efficiency when running the queries. Previous works have been able to provide quick responses by processing event data *in memory* [28]. Unfortunately, such approaches would not scale when dealing with large datasets, requiring thorough data preparation. By using the MapReduce paradigm WevQuery provides capabilities for quick hypothesis testing on large distributed datasets.

5 FUTURE WORK

WevQuery is currently a fully working system and, as future work, we aim to implement additional features that will further support designers. These features reduce the learning curve that is required to use the tool and increase the range of the possible queries. WevQuery has been developed within the context of an eScience project so future results will shed light on how scientists use collaborative tools. The mentioned future goals will be addressed through the following actions:

5.1 Abstracting Events

WevQuery has been designed to keep the required domain knowledge as low as possible. Unfortunately, there is still some necessary technical expertise to use all functionalities: designers need to understand the events within the sequences. For example, to extract all mouse click interaction, designers would need to look for `mousedown` and `mouseup` events. Nevertheless, removing these events would hinder the design of some complex queries. For example, designers may be interested in extracting dragging behaviours and would therefore need to include these mouse click events to define the start and end of the dragging action. The addition of new events to serve as abstractions would help simplify the use of the interface for those who are not familiar with Web conventions. In this case, a new event "mouse click" would be the equivalent of a `mousedown` followed by a `mouseup`. This has a straightforward solution in that WevQuery could provide a library of complex behaviours by default (conceived as templates) that designers could tailor to their specific needs.

In addition to these abstractions, we consider including higher level context options for the events. For example, the "scroll state" attribute can act as proxy to specify the relative scroll state as a context option for the event. The interface will offer designers a set of possible values for the event and the query will perform the necessary computations to determine its value. In this case, it will keep track of the scroll state of the page and its height to compute if the user is at the *top*, *center*, or *bottom* of the Web page.

5.2 Provide Support to Select Event Context

In WevQuery's current form, users with little or no programming skills only have to use the `ID` attribute of a particular DOM element to use any interface element in their hypotheses. For example, for the element `<img src='./pics/img5.png' id='search-button'>` a user would have to provide `'search-button'`, and this attribute is typically known by designers. Nevertheless, the need to know DOM elements' attributes restricts the selection of specific interface elements to experts. As an alternative, users will select the desired element with the mouse on the visual representation of the DOM. This sort of selection can be seen implemented in ad-blocking software for browsers.

5.3 Considering Unlimited Number of Occurrences

At the moment the occurrence of an event can be set to be a natural number. The possibility of allowing designers to query for an unlimited number of occurrences has been considered. However, this feature increases significantly the complexity of the transformation of the resulting XML into a database query, and additional requirements need to be defined to make its use effective. A possible use of this functionality would be querying for all hovering actions taking place within 10 seconds of a mouse click.

5.4 Support for Visual Analysis and Evaluation with Users

WevQuery provides designers with a JSON and CSV file containing all the patterns found as well as details on the matched events. Simplified results can be provided in a more accessible format through the visualisations of the results that designers could explore interactively. This way, user evaluations will be carried out to measure the effectiveness and the perceived complexity of WevQuery. Users' expertise in programming languages will be taken into account, to determine if programming experience influences on how WevQuery is used. The relation between the complexity and effectiveness of the constructed queries and the users' programming skill will also be investigated, to determine how to further support less skilled users.

6 CONCLUSION

We present WevQuery, an interactive query system for the analysis of fine-grained user interaction behaviour on Web systems. WevQuery enables the definition of complex interaction patterns by providing guidance on

the design of the queries, easing the effort and the necessary knowledge typically required to perform complex database queries. These queries are transformed and stored as XML files which allows for reuse and sharing, and are then transformed into scalable MapReduce queries. Consequently, WevQuery allows designers to test and iteratively refine their hypotheses about the behaviour of users on a Web system.

We illustrate the usefulness of WevQuery on two scenarios where two designers with differing opinions discuss the effect that certain aspects of the interface have on users' interaction. The scalability of the system was successfully tested employing low-level interaction data (5.7m events) captured from 2 445 users on a real website. WevQuery tackles the limitations of previous work in that it provides a flexible, extensible and scalable system to query log files containing user interface events.

REFERENCES

[1] David Akers, Matthew Simpson, Robin Jeffries, and Terry Winograd. 2009. Undo and erase events as indicators of usability problems. In *Proc. of the SIGCHI Conference on Human Factors in Computing Systems (CHI '09)*. 659–668. https://doi.org/10.1145/1518701.1518804

[2] James Allen. 1983. Maintaining Knowledge About Temporal Intervals. *Commun. ACM* 26, 11 (1983), 832–843. https://doi.org/10.1145/182.358434

[3] Morten Andreasen, Henrik Nielsen, Simon Schrøder, and Jan Stage. 2007. What Happened to Remote Usability Testing?: An Empirical Study of Three Methods. In *Proc. of the SIGCHI Conference on Human Factors in Computing Systems (CHI '07)*. 1405–1414. https://doi.org/10.1145/1240624.1240838

[4] Darko Anicic, Paul Fodor, Sebastian Rudolph, and Nenad Stojanovic. 2011. EP-SPARQL: A Unified Language for Event Processing and Stream Reasoning. In *Proc. of the 20th International Conference on World Wide Web (WWW '11)*. 635–644. https://doi.org/10.1145/1963405.1963495

[5] Aitor Apaolaza, Simon Harper, and Caroline Jay. 2013. Understanding Users in the Wild. In *Proc. of the 10th International Cross-Disciplinary Conference on Web Accessibility (W4A '13)*. Article 13, 13:1–13:4 pages. https://doi.org/10.1145/2461121.2461133

[6] Richard Atterer, Monika Wnuk, and Albrecht Schmidt. 2006. Knowing the user's every move: user activity tracking for website usability evaluation and implicit interaction. In *Proc. of the 15th International Conference on World Wide Web (WWW '06)*. 203–212. https://doi.org/10.1145/1135777.1135811

[7] Simon Breslav, Azam Khan, and Kasper Hornbæk. 2014. Mimic: Visual Analytics of Online Micro-interactions. In *Proc. of the 2014 International Working Conference on Advanced Visual Interfaces (AVI '14)*. 245–252. https://doi.org/10.1145/2598153.2598168

[8] Alan Cooper and others. 2004. *The inmates are running the asylum.* Sams Indianapolis, IN, USA.

[9] Gianpaolo Cugola and Alessandro Margara. 2010. TESLA: A Formally Defined Event Specification Language. In *Proc. of the Fourth ACM International Conference on Distributed Event-Based Systems (DEBS '10)*. 50–61. https://doi.org/10.1145/1827418.1827427

[10] Gianpaolo Cugola and Alessandro Margara. 2012. Processing Flows of Information: From Data Stream to Complex Event Processing. *ACM Comput. Surv.* 44, 3, Article 15 (2012), 15:1–15:62 pages. https://doi.org/10.1145/2187671.2187677

[11] Jeffrey Dean and Sanjay Ghemawat. 2008. MapReduce: Simplified Data Processing on Large Clusters. *Commun. ACM* 51, 1 (2008), 107–113. https://doi.org/10.1145/1327452.1327492

[12] Jerry Alan Fails, Amy Karlson, Layla Shahamat, and Ben Shneiderman. 2006. A visual interface for multivariate temporal data: Finding patterns of events across multiple histories. In *IEEE Symposium On Visual Analytics Science And Technology (VAST '06)*. 167–174.

[13] David Hilbert and David Redmiles. 2000. Extracting usability information from user interface events. *ACM Comput. Surv.* 32, 4 (2000), 384–421. https://doi.org/10.1145/371578.371593

[14] Jing Jin and Pedro Szekely. 2009. QueryMarvel: A visual query language for temporal patterns using comic strips. In *IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC '09)*. 207–214.

[15] Rosie Jones and Kristina Klinkner. 2008. Beyond the Session Timeout: Automatic Hierarchical Segmentation of Search Topics in Query Logs. In *Proc. of the 17th ACM Conference on Information and Knowledge Management (CIKM '08)*. 699–708. https://doi.org/10.1145/1458082.1458176

[16] Jonathan Lazar, Jinjuan Heidi Feng, and Harry Hochheiser. 2010. *Research Methods in Human-Computer Interaction.* John Wiley & Sons.

[17] Wanchun Li, Mary Jean Harrold, and Carsten Görg. 2010. Detecting user-visible failures in AJAX web applications by analyzing users' interaction behaviors. In *Proc. of the IEEE/ACM International Conference on Automated Software Engineering (ASE '10)*. 155–158. https://doi.org/10.1145/1858996.1859025

[18] Michael Nebeling, Maximilian Speicher, and Moira C. Norrie. 2013. CrowdStudy: General Toolkit for Crowdsourced Evaluation of Web Interfaces. In *Proc. of the 5th ACM SIGCHI Symposium on Engineering Interactive Computing Systems (EICS '13)*. 255–264. https://doi.org/10.1145/2494603.2480303

[19] Laila Paganelli and Fabio Paternò. 2002. Intelligent analysis of user interactions with web applications. In *Proc. of the 7th International Conference on Intelligent User Interfaces (IUI '02)*. 111–118. https://doi.org/10.1145/502716.502735

[20] Fabio Paternò, Antonio Schiavone, and Pierpaolo Pitardi. 2016. Timelines for Mobile Web Usability Evaluation. In *Proc. of the International Working Conference on Advanced Visual Interfaces (AVI '16)*. 88–91. https://doi.org/10.1145/2909132.2909272

[21] Ansgar Scherp, Thomas Franz, Carsten Saathoff, and Steffen Staab. 2009. F–a Model of Events Based on the Foundational Ontology Dolce+DnS Ultralight. In *Proc. of the Fifth International Conference on Knowledge Capture (K-CAP '09)*. 137–144. https://doi.org/10.1145/1597735.1597760

[22] Michael Terry, Matthew Kay, Brad Van Vugt, Brandon Slack, and Terry Park. 2008. Ingimp: introducing instrumentation to an end-user open source application. In *Proc. of the SIGCHI Conference on Human Factors in Computing Systems (CHI '08)*. 607–616. https://doi.org/10.1145/1357054.1357152

[23] Paul Thomas. 2014. Using Interaction Data to Explain Difficulty Navigating Online. *ACM Trans. Web* 8, 4, Article 24 (2014), 24:1–24:41 pages. https://doi.org/10.1145/2656343

[24] Chad Tossell, Philip Kortum, Ahmad Rahmati, Clayton Shepard, and Lin Zhong. 2012. Characterizing web use on smartphones. In *Proc. of the SIGCHI Conference on Human Factors in Computing Systems (CHI '12)*. 2769–2778. https://doi.org/10.1145/2208636.2208676

[25] Markel Vigo and Simon Harper. 2017. Real-time detection of navigation problems on the World 'Wild' Web. *International Journal of Human-Computer Studies* 101 (2017), 1–9. https://doi.org/10.1016/j.ijhcs.2016.12.002

[26] Eugene Webb, Donald Campbell, Richard Schwartz, and Lee Sechrest. 2000. *Unobtrusive Measures.* (second ed.).

[27] Harald Weinreich, Hartmut Obendorf, and Eelco Herder. 2006. Data Cleaning Methods for Client and Proxy Logs. In *Workshop on Logging Traces of Web Activity: The Mechanics of Data Collection.*

[28] Emanuel Zgraggen, Steven Drucker, Danyel Fisher, and Robert DeLine. 2015. (s|qu)Eries: Visual Regular Expressions for Querying and Exploring Event Sequences. In *Proc. of the SIGCHI Conference on Human Factors in Computing Systems (CHI '15)*. 2683–2692. https://doi.org/10.1145/2702123.2702262