Mining Behavior Models from User-Intensive Web Applications*

Carlo Ghezzi
Politecnico di Milano, Italy
DeepSE Group at DEIB
carlo.ghezzi@polimi.it

Mauro Pezzè
University of Lugano, Switzerland
mauro.pezzè@usi.ch

Michele Sama
Head of cloud and data
Touchtype Ltd, UK
michele@swiftkey.net

Giordano Tamburrelli
University of Lugano, Switzerland
giordano.tamburrelli@usi.ch

ABSTRACT

Many modern user-intensive applications, such as Web applications, must satisfy the interaction requirements of thousands if not millions of users, which can be hardly fully understood at design time. Designing applications that meet user behaviors, by efficiently supporting the prevalent navigation patterns, and evolving with them requires new approaches that go beyond classic software engineering solutions. We present a novel approach that automates the acquisition of user-interaction requirements in an incremental and reflective way. Our solution builds upon inferring a set of probabilistic Markov models of the users' navigational behaviors, dynamically extracted from the interaction history given in the form of a log file. We annotate and analyze the inferred models to verify quantitative properties by means of probabilistic model checking. The paper investigates the advantages of the approach referring to a Web application currently in use.

Categories and Subject Descriptors

D.2.4 [**Software Engineering**]: Software/Program Verification—*Model Checking*; H.3.4 [**Information Storage and Retrieval**]: Systems and Software—*User profiles*

General Terms

Design, Measurement, Verification

Keywords

Web Application, Log Analysis, User Profiles, Markov Chains, Probabilistic Model Checking

*This research has been funded by the EU: Programme IDEAS-ERC, Prj. 227977-SMScom and by a Marie Curie IEF within the 7^{th} European Community Framework Programme: Prj. 302648-RunMore

ICSE '14, May 31 - June 7, 2014, Hyderabad, India

1. INTRODUCTION

A key distinguishing feature of user-intensive software, and in particular Web applications, is the heavy dependence on the interactions with many users, who approach the applications with different and evolving needs, attitudes, navigation profiles, preferences, and even idiosyncrasies[1], which generate different navigation profiles. Knowing and predicting the different user behaviors are crucial factors that may directly affect the success of the application. Underestimating the importance of these factors may lead to technical as well as non-technical failures that may involve substantial economic losses. For example, an inadequate or distorted knowledge of users' navigation preferences may lead to Web applications characterized by an unsatisfactory user experience with consequent loss of customers and revenues.

Unfortunately the presence of a huge number of users with different and evolving behaviors make it almost impossible to accurately predict and model all of them, and to design applications that can answer all possible needs. Moreover, the population of users is seldom homogenous and, typically, several *classes of users* with distinct user behaviors coexist at the same time. In addition, no matter how well they are initially captured, user behaviors change over time. This leads to the need for learning and refining our understanding of how users interact with the system and to the need for speculating on the inferred knowledge to drive the progressive system maintenance, adaptation, and customization.

The mainstream approach towards capturing the user behaviors consists of *monitoring* the usage of the system and subsequently *mining* possible interaction patterns [35]. Some existing solutions instrument Web pages to track users' navigation actions – for instance Google Analytics [1] – while others analyze log files as discussed by Facca and Lanzi [12]. Current solutions suffer from several limitations. Some approaches lack generality, for example, they need to infer users' navigational profiles to support specific actions, such as run-time link prediction [31] or data caching [39]. Being tailored to specific tasks, these solutions provide little support from a general software engineering perspective. On the other hand, the general frameworks simply return a set of statistics or patterns that are useful to understand the preferences of system's users but cannot be directly used to

[1]We collectively identify these factors under the term *user behavior*, or simply *behavior* when the context is clear.

evaluate software engineering decisions.

This paper describes a framework, called BEAR[2], that overcomes these limitations supporting software engineers in maintaining and adapting existing user-intensive Web applications. The proposed approach focuses on REST[3] architectures and analyses the log file of the server in which the application under analysis has been deployed. By analyzing the log file, BEAR infers a set of Markov models that cluster similar users in classes and capture their behaviors probabilistically. Developers can decorate the inferred models with *rewards* that represent the causal connection among user navigation actions and technical as well as non-technical aspects of the application under analysis. An ad-hoc analysis engine mines the models to gather valuable insights about the users' behaviors and the relation among users and the entities modeled by the rewards. The analysis engine relies on *Probabilistic model checking* [4] to formally verify quantitative properties of the behavior of the users captured by the models. For instance, the analysis engine may compute the probability that a user, who enters the Web application from a certain link, navigates across one or more specific paths and reaches a given target page. The analyses supported by our approach may focus on the whole population of users, on a specific class of users in isolation, or may compare different classes.

BEAR can be viewed as an approach to facilitate and automate the acquisition of user-interaction requirements in an incremental and reflective way that supports many aspects of the application maintenance and evolution. The proposed solution brings some key advantages with respect to existing competing alternatives. The inferred models, together with rewards, represent a general abstraction that can be used to analyze technical as well as non-technical and domain specific aspects of the application. Probabilistic model checking and the incremental inference process allow the progressive and automatic analysis of large and complex models as soon as new data become available in the log file.

To the best of our knowledge, this paper contributes to current research in requirements and design of user-intensive software in three distinct ways:

1. It is the first approach that investigates the use of probabilistic model checking to formally verify quantitative properties of users' behaviors in Web applications.

2. It investigates a novel approach to capture user behaviors with an inference algorithm and a model-based technique specifically conceived for REST architectures.

3. By introducing rewards, it provides a new way to reason about the relationships between the interaction of users and several aspects of the application under analysis.

The remainder of the paper is organized as follows. Section 2 overviews the BEAR approach. Section 3 introduces the running case study we use throughout the paper to exemplify and validate the approach. Section 4 provides a detailed description of the approach, while Section 5 illustrates some relevant scenarios of our case study that exemplify the potential applications of BEAR. Section 6 discusses performance and scalability of the approach. Section 7 discusses related work. Section 8 summarizes the main contributions of the paper, and illustrates the ongoing research work.

[2] *BEhavioral Analysis of REST applications.*
[3] *REpresentational State Transfer [13].*

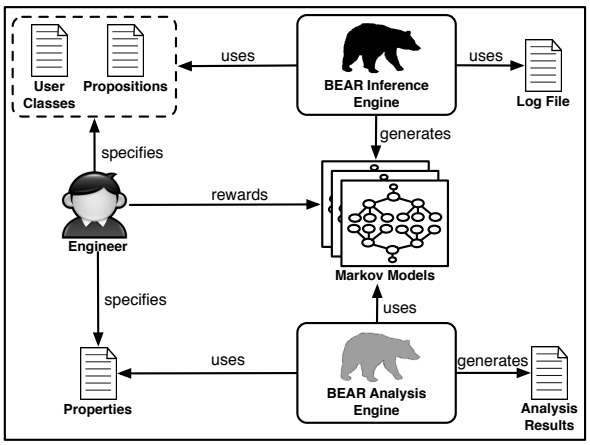


Figure 1: The BEAR approach

2. THE BEAR APPROACH

In this section we present the BEAR approach. Although the approach is applicable to different kinds of systems and logs, in this paper, we refer to user interactions with remote services implemented according to a REST architectural style [13, 37]. REST is an increasingly popular architectural style, in which requests and responses flow through the *stateless* transfer of resources via URLs that uniquely identify the state of the conversation between clients and servers.

BEAR infers the user behaviors by deriving Markov models from log files, and quantitatively verifies properties of the interaction patterns by analyzing the inferred models using probabilistic model checking. The verification provides the core information to refactor or customize the application according to emerging user behaviors. BEAR is articulated in six main steps that we introduce below referring to the diagram in Figure 1, and discuss in detail in Section 4:

1. *Identifying the atomic propositions*: The designers give semantics to the URLs occurring in the log file by means of a set of atomic propositions that denote the relevant user actions. This is a necessary setup phase through which the designer identifies the actions relevant for the analysis of the application. BEAR automatically clusters the entries of the log that represent URLs into groups univocally identified by sets of propositions.

2. *Identifying user classes*: The designers characterize the population of users by identifying a set of relevant features to cluster them into distinct classes. For instance, they may use the feature *user-agent* to discriminate users depending on the browser they rely on, or depending on the device they use (mobile vs. desktop users).

3. *Inferring the models*: The BEAR *inference engine* analyses the log file of the application and infers a set of discrete time Markov chains (DTMCs) [4]. DTMCs are finite state automata augmented with probabilities: each state is characterized by a discrete probability distribution that regulates the outgoing transitions. The inference engine generates an independent DTMC for each user class.

4. *Annotating the models with rewards*: The designers may provide information by annotating the states of the models with numerical values that represent *rewards* [22]. Rewards indicate the impact of the state on some metrics of interest. The annotations are optional and refer to the set of atomic propositions introduced in the second step.

5. *Specifying the properties of the interaction patterns*: The designers formally specify the properties of interest for the user-intensive system. The properties may predicate on the probability that users may follow a certain navigational pattern, or may predicate on the rewards.

6. *Analyzing the models*: The BEAR *analysis engine* quantitatively evaluates the formal properties against the Markov models, and produces either numerical or boolean results, depending on the nature of the properties. The obtained results provide insights on the behaviors of the users and on the impact of such behaviors on the rewards in the models. These insights guide designers in refactoring or customizing the application under analysis.

3. THE REAL-ESTATE EXAMPLE

We introduce the real-world REST application we use throughout the paper to illustrate and validate the BEAR approach. The running example has been provided by an IT consultancy company and is currently in use. We chose it because it balances generality and simplicity: the application is general enough to include the main aspects that characterize the requirements in the user-intensive REST domain, and simple enough to illustrate the approach. During our study we had full access to both the application and the log files, and we conducted our experiments on the data collected during the normal operation of the application. For confidentiality reasons we anonymize the application selected as a case-study by using the pseudonym *findyourhouse.com*. Through the findyourhouse.com website, users may either browse the housing announcements divided by category, or search for the available offers with a proprietary search engine accessible from every page of the website. If users find an announcement of interest, they can contact a sales agent for additional details and schedule a visit to the property of interest. The website is composed of several pages and resources identified by the URLs summarized in Table 1 (the meaning of the third column will be discussed later and can be ignored at this stage). The table contains samples of the relevant URLs and provides a summary of the URLs through a prefix. It ignores URLs that are not relevant for the experiments, for example the URL pointing to the `robots.txt` file used by Web spiders. Some of the URLs are simple, for example `/home/`, while others are parametrized, for example `/anncs/sales/?page=<n>` or have a parametric structure, for example `/anncs/sales/<id>/`.

Even for a relatively simple REST application like the one illustrated in this section, information about the behavior of the final users is crucial for engineering an application that meets users' different and evolving browsing requirements and tries to increase the economic value of the service.

4. THE DETAILS OF THE APPROACH

BEAR is grounded on a simple basic assumption, discussed here before presenting the six steps of the approach in details. The input to BEAR is a log file structured as a list of rows that record the interactions between the users and the Web server of the application under analysis. Each row represents a request of a Web resource issued by a client. Hereafter we use the terms *row* and *request* interchangeably. We assume that the rows contain the following common data: the IP address of the user who issued the request to the server, a timestamp that represents the time of the request, the user-agent, and the requested URL. These assumptions correspond to the information provided by the log files compliant to the Common Log Format (CLF) adopted by many popular Web servers, such as the Apache Web server[4].

4.1 Identifying the Atomic Propositions

In the first step of the approach BEAR associates semantics to the rows of the log file by means of a set of atomic propositions ($\mathcal{AP}$) that indicate what can be assumed as valid when a certain entry in the log file is found. For example, the proposition *homepage* is associated to a row in the log file to indicate that the request corresponding to that row has led the application to the home page.

The atomic propositions to be associated to the log entries are encoded as code fragments, called *filters*. The application designers declare the filters as methods decorated with the annotation **@BearFilter**. A filter may be parameterized with a regular expressions, to restrict its application to the URLs that match the expression. The BEAR engine scans the log file, invokes the filters with matching parameters on each row it analyses, and associates the propositions returned by the filters to the log file entries.

The third column in Table 1 shows some atomic propositions associated to URLs of the findyourhouse.com application. Listing 1 shows some examples of filters. The first filter associates the proposition *homepage* to the homepage URL. The second filter associates the proposition *sales_anncs* to the URLs that matches the regular expression in its annotation, and indicates that the URLs are related to sales announcements. The third filter associates the propositions *login_success* and *login_fail* to URLs that correspond to login attempts, depending on the HTTP status code of the request. The fourth filter associates the propositions *control_panel* to the pages of the control panel. The BEAR inference engine associates propositions only to relevant URLs, which match some regular expressions in the filters. Rows that correspond to URLs associated to secondary resources that are not crucial to capturing relevant aspects of the users behavior are not labeled with any proposition. For example, usually filters do not match URLs that represent CSS or Javascript resources that are not relevant to user navigation actions, since they are automatically requested by browsers, and thus are not labeled with propositions. Filters may generate propositions dynamically according to information available in the URL or the application database. For example, a filter may use the announcement ID extracted from the URL to retrieve the location of the sales announcement from the application database, and may use the location to build a proposition dynamically.

Filters, propositions and regular expressions are flexible tools that application designers use to characterize the rows in the log file, exploiting both application and domain specific knowledge.

[4] `http://httpd.apache.org/docs/2.4/logs.html`

Table 1: The relevant URLs of findyourhouse.com

URL	Description	Atomic Propositions
/home/	Homepage of findyourhouse.com	*homepage*
/anncs/sales/	The first page that shows the sales announcements.	*sales_page*, *page_1*
/anncs/sales/?page=<n>	The n^{th} page that shows sales announcements, <n> is an integer index.	*sales_page*, *page_n*
/anncs/sales/<id>/	Detailed view of the sales announcement identified by the string <id>.	*sales_anncs*
/anncs/renting/	The first page that shows the renting announcements.	*renting_page_1*
/anncs/renting/?page=<n>	The n^{th} page that shows renting announcements, <n> is an integer index.	*renting_page*, *page_n*
/anncs/renting/<id>/	Detailed view of the renting announcement identified by the string <id>.	*renting_anncs*
/search/	Page containing the results of a search submitted through the search engine.	*search*
/admin/.../	Website's control panel that allows to publish, edit or delete announcements.	*control_panel*
/admin/login/	Login page that allows to access the control panel.	*login_success* or *login_fail*
/contacts/	URL with the form to contact a sales agent.	*contacts*
/contacts/submit/	URL that indicates that the form used to contact the agency has been submitted.	*contacts_requested*
/contacts/tou/	Page that describes the website terms of use.	*tou*
/media/.../	URLs with this prefix refer to images, CSS and Javascript resources.	-

```
@BearFilter(regex="^/home/$")
public static Proposition void filterRenting(LogLine line){
   return new Proposition("homepage");
}

@BearFilter(regex="^/anncs/sales/(\w+)/$")
public static Proposition void filterSales(LogLine line){
   return new Proposition("sales_anncs");
}

@BearFilter(regex="^/admin/login/$")
public static Proposition void filterLogin(LogLine line){
   if(logLine.getHTTPStatusCode == "302")
      return new Proposition("login_success");
   else
      return new Proposition("login_fail");
}

@BearFilter(regex="^/admin/edit/$")
public static Proposition void filterAdmin(LogLine line){
   return new Proposition("control_panel");
}
```

Listing 1: Some examples of filters

4.2 Identifying the User Classes

In this step of the approach, the designer may define a set of user classes relevant for the application under analysis. A string in the format *(name="value")* defines each user class. The BEAR inference engine uses code fragments called *classifiers* decorated with the annotation **@BearClassifier** to specify classes of users. The BEAR engine scans the log file, invokes the classifiers on each row, and associates the user classes returned by the classifiers to the log file entries. By default, BEAR comes with two classifiers that extract the user-agent and the user's location obtained geolocating the IP address. For instance a row may by associated with the following user classes:

$$\{(userAgent = \text{“}Mozilla/5.0...\text{”}), (location = \text{“}Boston\text{”})\}$$

Classifiers represent a flexible and extensible tool to map rows in the log into classes. Notice that, as shown in the example above, each user may belong to multiple classes.

By adding classifiers the designers can classify the users into application or domain specific classes exploiting additional information that may be stored in customized log files. For example designers may classify users predicating on their operating system, the HTTP referrer, the user's time zone, etc. For the sake of readability in this paper we refer only to the default classifiers, even if all the concepts and examples we discuss here apply seamlessly to more complex and application specific classifiers.

4.3 Inferring the Model

Given the set of atomic propositions $\mathcal{AP}$ and the user classes defined through filters and classifiers, respectively, the BEAR inference engine infers a set of discrete time Markov chains (DTMCs) [4] that represent the users' behaviors. The inference process works sequentially and incrementally on the log file as a data stream. Once a log entry is processed, it may be discarded. Thus the process works efficiently both on-line and off-line, and works both for legacy applications for which log data have been collected and for newly deployed applications.

The inference engine generates an independent DTMC for each user class defined by the classifiers. For the sake of readability we first define the inference of a single generic DTMC model, and then extend the inference algorithm to multiple DTMCs later in this section. In this paper we derive DTMCs augmented with rewards [3], defined as follows.

A DTMC is a tuple $\langle \mathcal{S}, \mathcal{P}, \mathcal{L}, \rho \rangle$ where:

- $\mathcal{S}$: is a non empty finite set of states, and $s_0 \in S$ is the initial state;
- $\mathcal{P} : \mathcal{S} \times \mathcal{S} \to [0, 1]$ is a stochastic matrix that represents the probabilistic edges that connect the states in $\mathcal{S}$. An element $\mathcal{P}(s_i, s_j)$ represents the probability that the next state will be s_j given that the current state is s_i;
- $\mathcal{L} : \mathcal{S} \to 2^{\mathcal{AP}}$ is a labeling function that associates each state with a set of atomic propositions $\mathcal{AP}$.
- $\rho : S \to \mathbb{R}_{\geq 0}$ is a *state reward* function that assigns a non-negative number to each state. Rewards are non-negative values that quantify the benefit (or disadvantage) of being in a specific state.

As for the DTMCs we infer, the set $\mathcal{AP}$ consists of the propositions identified by filters, as described in Section 4.1. In addition, no two different states can be associated with the same set of atomic propositions; i.e., the set of atomic propositions associated with a state univocally identify it. The inference process starts from an initial DTMC, characterized by:

- A set of states $\mathcal{S}$ that contains the initial state s_0 and a sink state e that models users who leave the system;
- A stochastic matrix $\mathcal{P}$ with only one non-zero element: $\mathcal{P}(e, e) = 1$;
- A labeling function $\mathcal{L}$ defined as follows:

```
IP              TIMESTAMP                URL
1.1.1.1 - [20/Dec/2011:15:35:02] - /home/
2.2.2.2 - [20/Dec/2011:15:35:07] - /admin/login/
1.1.1.1 - [20/Dec/2011:15:35:12] - /anncs/sales/1756/
2.2.2.2 - [20/Dec/2011:15:35:19] - /admin/edit/
```

Listing 2: Log file excerpt (the listing does not report the user-agents)

$$\mathcal{L}(s) = \begin{cases} \{start\} & \text{if } s = s_0 \\ \{end\} & \text{if } s = e \\ \emptyset & otherwise \end{cases}$$

- A reward function ρ that assigns 0 to all the states in $\mathcal{S}$. Non-null rewards can be added as discussed in Section 4.4.

This initial DTMC is shown in Figure 2(a). The inference engine builds the DTMC incrementally by processing the rows of the log file and inferring *transitions* between states in $\mathcal{S}$. The engine processes each row in four steps:

1. *Extracting the destination state:* The inference engine examines the propositions associated by the filters with the current row r. The engine ignores the rows associated with an empty set of propositions, since they represent transitions irrelevant in our context. The engine associates r with a destination state $d \in \mathcal{S}$ such that $\mathcal{L}(d) = l$, where l is the set of propositions associated with r. If d does not belong to $\mathcal{S}$ yet, the inference engine adds the new state d to $\mathcal{S}$ and updates the labeling function $\mathcal{L}$ accordingly.

2. *Extracting the user identifier:* The engine assumes that distinct IP addresses correspond to different users, thus assigns a unique identifier to each new IP address extracted from the rows in the log file. In normal operational conditions, IP addresses may not be uniquely associated to users, and it may happen that different users that browse the system in different occasions may have the same IP address. To cope with this scenario, the inference engine refers to the timestamps of the transitions, and assumes that requests issued from the same IP address with significantly different timestamps are issued by different users (i.e., they are given distinct user identifiers). The minimum temporal distance between timestamps to consider two requests with the same IP address as issued by distinct users is a parameter, called *userwindow*.

3. *Extracting the source state:* If the previous step assigned a new user identifier to r, the engine assumes that r is the first interaction of the user with the system, and associates the initial state s_0 as the source state. If the interaction r comes from an existing user (a known IP address), the inference engine retrieves the destination state of the most recently processed row characterized by the same IP address, and assigns such state as the source state of r.

4. *Computing the probabilities:* The engine uses the transitions extracted from the log file to update two sets of counters that are initially set to zero: a set of counters $c_{i,j}$ for each pair of states $(s_i,\, s_j) \in \mathcal{S}\times\mathcal{S}$, and a set of counters t_i for each state $s_i \in S$. The engine increments both the counter $c_{i,j}$ for each transition from state s_i to s_j and the counter t_i for each transition whose source state is s_i, independent of its destination state. The counter t_i represents the number of times the users exited state s_i, while counter $c_{i,j}$ represents the number of times the users moved from state s_i to state s_j. The inference engine updates the counters for each row in the log file that corresponds to a transition in the model, and uses these counters to compute the (i, j) entry of the stochastic matrix $\mathcal{P}$ that represents the probability of traversing the edge from state s_i to state s_j, by computing the following frequency:

$$\mathcal{P}(s_i,\, s_j) = \frac{c_{i,j}}{t_i}$$

for all pairs of states s_i and s_j. The probability $\mathcal{P}(s_i,\, s_j)$ is computed as the ratio between the number of traversals of the transitions from state s_i to s_j and the total number of traversals of the transitions exiting state s_i, and corresponds to the *maximum likelihood estimator* for $\mathcal{P}(s_i,\, s_j)$ [11]. The probabilities can be recomputed incrementally after adding any number of transitions or states to the DTMC.

Figure 2 illustrates the inference process described so far referring to the log file in Listing 2 and the filters shown in Listing 1. The log entries in Listing 2 have been excerpted from the interactions of two users with the findyourhouse.com application. The BEAR engine starts from the initial DTMC shown in Figure 2(a), and proceeds incrementally through the log file. BEAR associates the first row with the proposition *homepage*. Since $\mathcal{S}$ does not contains any state s such that $\mathcal{L}(s) = \{homepage\}$, the engine adds a new state s_1 to $\mathcal{S}$, and extends the labeling function with $\mathcal{L}(s_1) = \{homepage\}$. Being this the first transition in the log, the IP address has not been already encountered, thus the engine considers state s_0 as the source state for the inferred transition ($\langle s_0, s_1\rangle$). The engine increments the counters t_0 and $c_{0,1}$, and consequently sets $\mathcal{P}(s_0,\, s_1)$ to 1. Figure 2(b) shows the resulting DTMC.

The second row corresponds to a successful login and BEAR associates it with the set of propositions {*login_success*, *control_panel*}. The engine associates this row with the new destination state s_2 and updates the labeling function: $\mathcal{L}(s_2)$={*login_success*,*control_panel*}. The IP address is new, and thus the engine creates a new user identifier and associates the transition with the initial state s_0 yielding to a transition $\langle s_0, s_2\rangle$. The engine increments the counters related to the new transition: $t_0 = 2$ and $c_{0,2} = 1$, and sets $\mathcal{P}(s_0,\, s_2)$ and $\mathcal{P}(s_0,\, s_1)$ to 0.5. Figure 2(c) shows the resulting DTMC.

The engine processes the third and fourth row in a similar way. It associates the third row with a new destination state s_3 such that $\mathcal{L}(s_3)$={*sales_anncs*}. Since the row is associated with the same IP address of the first one, the source state corresponds to the destination state of the last transition generated by this user (s_1) and results in the transition $\langle s_1, s_3\rangle$. The engine increments the counters t_1 and $c_{1,3}$, and consequently sets $\mathcal{P}(s_1,\, s_3) = 1$.

When processing the fourth row, BEAR generates a transition from s_2 (the destination state of the second transition that corresponds to the same IP address) to s_4 with $\mathcal{L}(s_4) = \{control\_panel\}$. The engine increments the counters t_2 and $c_{2,4}$, and sets $\mathcal{P}(s_2, s_4) = 1$. Figure 2(d) shows the resulting DTMC. When the *userwindow* timeout for a certain IP address expires, the engine assumes that the user associated with that address left the system. As a consequence, a later request from the same address is considered as issued by a new user and the source state is s_0. As soon as

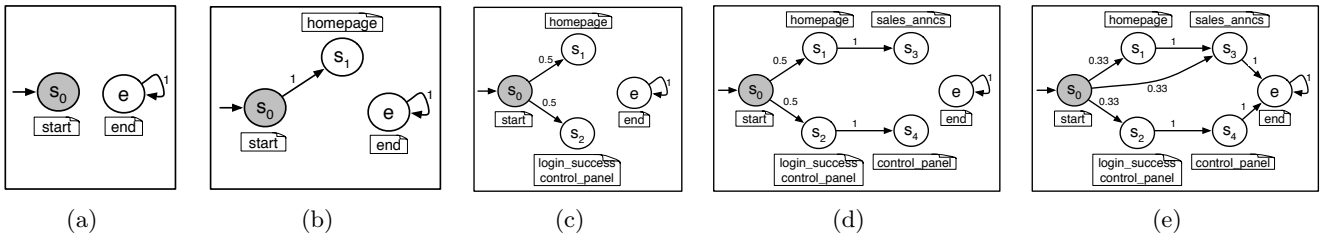


Figure 2: DTMC inference process

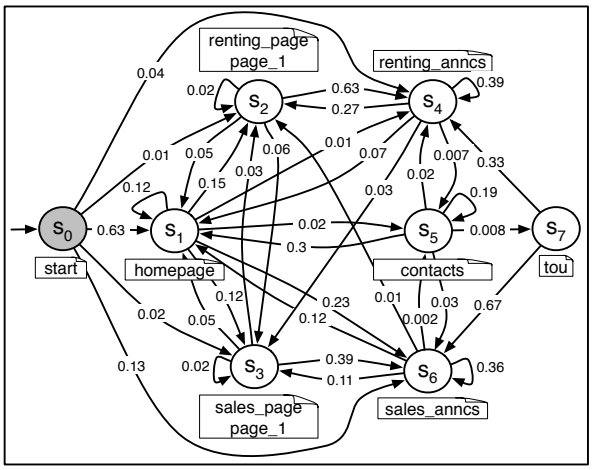


Figure 3: Partial DTMC of findyourhouse.com

a the timeout expires, the engine generates a new transition that leads from the destination state of the last transition issued by the expired IP address towards the final state e. If for example we assume that the *userwindow* timeout expires for both the considered users, the engine generates the transitions $\langle s_4, e\rangle$ and $\langle s_3, e\rangle$, and updates the corresponding counters. A new interaction occurring after the timeout that leads to state s_3 and involves the IP address of one of previous requests generates a new user and a transition $\langle s_0, s_3\rangle$. Figure 2(e) shows the resulting DTMC.

By processing the log file of the findyourhouse.com application with a set of filters that match the propositions shown in the third column of Table 1, the BEAR engine produces a DTMC with 23 states and 214 transitions. Figure 3 shows a subset of the generated model. Being the model a subset of the complete DTMC, the probability of the transitions exiting a state may not sum to one.

The obtained DTMC captures the behaviors of all users regardless of the class they belong to. In general, the BEAR inference engine infers a DTMC for each class of users. For instance, if a log file contains entries characterized by five different user-agents and ten different locations, the inference engine infers fifteen DTMCs, one for each class of user-agents and one for each class of user locations. Each log entry processed by the engine contributes uniquely to the inference of the DTMCs associated to its user classes. For instance a row associated with the user classes exemplified in Section 4.2 contributes to infer a DTMC associated to the class: ($userAgent$ = "$Mozilla/5.0$...") and a DTMC associated to the class: ($location$ = "$Boston$"). In this setting, each inferred DTMC captures the behavior of a specific class of users. The classifiers identified by the application designer directly affect the number of inferred DTMCs and their level of abstraction.

4.4 Annotating the Models

Rewards are non-negative values the designer can associate with propositions to model benefits or losses. The BEAR engine uses the rewards associated with propositions in the setup phase to automatically annotate the states of all inferred DTMCs.

In the absence of rewards, BEAR returns a set of models with default value *zero* for the reward function ρ of the states. In the presence of rewards, the BEAR inference engine computes the reward of the states as the sum of the rewards of the propositions associated with the states. Let us consider the DTMC in Figure 3, and let us assume that the designer assigns rewards 2 and 3 to the propositions *sales_page* and *page_1*, respectively, then the BEAR engine assigns 5 to $\rho(s_3)$.

Rewards are a general purpose and abstract tool to augment the inferred DTMCs with domain specific metrics of interests. Designers can use rewards to capture both technical and non-technical metrics of interest. They assign rewards to propositions in the setup phase, and do not need to know the structure and the number of the inferred models, which are inferred only later while monitoring the application behavior.

Here we illustrate the use of rewards with an example related to a refactoring of the findyourhouse.com application, which aims at increasing the number of announcements displayed to the customers, and thus gives special recognition to the states that display a large number of announcements. This can be easily achieved by annotating the propositions with rewards that depend on the number of announcements they are related with. They associate reward 6 with the proposition *homepage*, because homepage displays six announcements. Likewise, they associate reward 9 with both the propositions *page_1* and *page_n*, because they contain nine announcements, and so on. The BEAR engine computes the reward functions of the states of the inferred DTMCs accordingly. For instance in the DTMC in Figure 3, $\rho(s_1) = 6$ since s_1 is associated only to proposition *homepage*, $\rho(s_2) = 9$ and $\rho(s_3) = 9$.

4.5 Specifying the Properties

In this step, the application designers define the properties of interest using Probabilistic Computation Tree Logic (PCTL) augmented with rewards [17, 22]. PCTL with rewards is

a probabilistic branching-time temporal logic based on the classic CTL logic [4] that predicates on a state of a Markov process. Hereafter we provide a formal definition of this logic through the following recursive grammatical rules:

$$\phi ::= true \mid a \mid \phi \ \wedge \ \phi \mid \neg \ \phi \mid \mathcal{P}_{\bowtie p} \ (\psi) \mid \mathcal{R}_{\bowtie r} \ (\Theta)$$
$$\psi ::= \mathcal{X} \ \phi \mid \phi \, \mathcal{U}^{\leq t} \ \phi$$
$$\Theta ::= \mathcal{I}^{=k} \mid \mathcal{C}^{\leq k} \mid \ \Diamond \phi$$

where $p \in [0,1]$, $\bowtie \in \{<, \leq, >, \geq\}$, $t \in \mathbb{N} \cup \{\infty\}$, $r \in \mathbb{R}_{\geq 0}$, and $k \in \mathbb{Z}_{\geq 0}$, and a represents an atomic proposition. The operator $\mathcal{R}_{\bowtie r}$ (Θ) supports the specification of properties that predicate over rewards. Let us first discuss the semantics of basic PCTL ignoring the reward operator. Formulae originated by the axiom ϕ are called *state formulae*; those originated by ψ are instead called *path formulae*. The semantics of a state formula is defined as follows:

$$\begin{array}{lll} s & \models true & \\ s & \models a & \text{iff} \quad a \in \mathcal{L}(s) \\ s & \models \neg\phi & \text{iff} \quad s \not\models \phi \\ s & \models \phi_1 \wedge \phi_2 & \text{iff} \quad s \models \phi_1 \ and \ s \models \phi_2 \\ s & \models \mathcal{P}_{\bowtie p}(\psi) & \text{iff} \quad Pr(\pi \models \psi | \pi[0] = s) \bowtie p \end{array}$$

where $Pr(\pi \models \psi | \pi[0] = s)$ is the probability that a path originating in s satisfies ψ. A path π originating in s satisfies a path formula ψ according to the following rules:

$$\begin{array}{lll} \pi & \models \mathcal{X}\phi & \text{iff} \quad \pi[1] \models \phi \\ \pi & \models \phi_1 \mathcal{U}^{\leq t} \phi_2 & \text{iff} \quad \exists \, 0 \leq j \leq t \quad (\pi[j] \models \phi_2 \wedge \\ & & \quad (\forall 0 \leq k < j \, \pi[k] \models \phi_1)) \end{array}$$

Let us now discuss intuitively the reward operator $\mathcal{R}_{\bowtie r}$ (Θ):

- $\mathcal{R}_{\bowtie r}(\mathcal{I}^{=k})$ is true in state s if the expected state reward to be gained in the state entered at step k along the paths originating in s meets the bound $\bowtie r$.
- $\mathcal{R}_{\bowtie r}(\mathcal{C}^{\leq k})$ is true in state s if, from s, the expected reward *cumulated* after k steps meets the bound $\bowtie r$.
- $\mathcal{R}_{\bowtie r}(\Diamond\phi)$ is true in state s if, from s, the expected reward cumulated before reaching a state where ϕ holds meets the bound $\bowtie r$.

The BEAR analysis engine processes PCTL properties relying on the PRISM [23] probabilistic model checker[5]. More precisely, a BEAR property is composed of a preamble in curly brackets and a PCTL expression. The preamble defines the *scope* of the property, while the PCTL expression specifies the formula to be verified with the model checker.

The scope of the property is expressed as a string in the from *name* = "*regex*" that identifies the user classes the PCTL formula refers to. More precisely, the first part of the scope (i.e., *name*) is the name of a user class as specified with the classifiers, for instance, a *userAgent*, while the second part (i.e., *regex*) corresponds to a regular expression. The property verification is restricted to the users that belong to the specified class, and that matches the regular expression in the scope of the property.

As an example, the designers of findyourhouse.com who are interested in the probability that users with a certain browser, for instance Mozilla, will eventually contact a sales agent can use the following property:

$$\{userAgent = \text{“}(.*)Mozilla(.*)\text{”}\}\mathcal{P}_{=?}[\, F \, contact\_requested \,] \quad (1)$$

As another example, the application owners who are interested in the number of announcements displayed to mobile users, for example, Android or iOS users, through all the states up to the final state can use the following property:

$$\{userAgent = \text{“}(.*)(Android|iOS)(.*)\text{”}\}\mathcal{R}_{=?}[F \, end] \quad (2)$$

that refers to the reward function that associates the number of displayed announcements to each state as exemplified in the previous paragraph. The PCTL properties depend only on the propositions, and designers can specify them without knowing the structure of the inferred models in terms of states or transitions. Designers can also express properties in structured english that can be automatically translated into PCTL properties [16]. Indeed, designers can be completely agnostic of the complexity of the formal tools such as the probabilistic model checking engine and PCTL used by the BEAR analysis engine.

4.6 Analyzing the Models

The process is completed with the analysis of the properties specified by the application designer. The BEAR engine exploits the scope in the preamble of the properties to identify the set of relevant DTMCs among the inferred models. As discussed in Section 4.3, the BEAR inference engine generates a distinct DTMC for each user class.

For each property to be processed, the analysis engine selects the DTMCs associated with the user classes that match the regular expression indicated in its scope. For example when analyzing property (1), the analysis engine selects the DTMCs associated to the Mozilla user-agents. The BEAR analysis engine selects one or more DTMCs depending on the user agents contained in the log entries processed during the inference process. For instance, it may select the DTMC associated to the user-agents of Mozilla 5.0 as well as the DTMCs associated to user-agents of different versions of Mozilla since they all match the regular expression in the property's scope.

If the scope of the property selects multiple DTMCs, the BEAR analysis engine merges them and *synthesizes* a single DTMC as described hereafter. Let $\{\mathcal{D}_1, \mathcal{D}_2, \ldots, \mathcal{D}_n\}$ be the set of DTMCs selected according to the scope of a property p, where $\mathcal{D}_k = \langle \mathcal{S}_k, \mathcal{P}_k, \mathcal{L}_k, \rho_k \rangle$, for each $1 \leq k \leq n$, and let u_k be the user class associated to $\mathcal{D}_k$, for each $1 \leq k \leq n$, the merging procedure produces a new DTMC $\mathcal{T} = \langle \mathcal{S}_\mathcal{T}, \mathcal{P}_\mathcal{T}, \mathcal{L}_\mathcal{T}, \rho_\mathcal{T} \rangle$ where:

- The set of states is the union of the sets of states of the input DTMCs:

$$\mathcal{S}_\mathcal{T} = \bigcup_{1 \leq k \leq n} \mathcal{S}_k$$

  States of different input DTMCs may correspond to the same state of the merged DTMC since, by construction, two states are equal if and only if they are characterized by the same propositions, see Section 4.3.

[5]We present the PTCL properties in the PRISM syntax.

- The transition probabilities are computed according to the law of total probability [11]:

$$\mathcal{P}_{\mathcal{T}}(s_i,\, s_j) = \sum_{1 \le k \le n} \mathcal{P}_k(s_i,\, s_j) \times \mathcal{P}_i(u_k)$$

  where $\mathcal{P}_i(u_k)$ corresponds to the probability for a user (associated to the selected user classes) that exited state s_i to belong to the specific user class u_k.
- The labeling function is computed as:

$$\mathcal{L}_{\mathcal{T}}(s) = \mathcal{L}_k(s) \text{ if } s \in S_k$$

  The readers should notice that if a state belongs to more than one DTMC, the labeling function of all the DTMCs returns the same value by construction.
- The reward function is computed as:

$$\rho_{\mathcal{T}}(s) = \rho_k(s) \text{ if } s \in S_k$$

  The readers should notice that if a state belongs to more than one DTMC, the reward function of all the DTMCs returns the same value by construction. In addition the distribution of rewards is independent from the merging process since, as explained in Section 4.4, they are defined for propositions.

The synthesized DTMC captures the behaviors of several user classes. For example, the DTMC synthesized for property (1) captures the behavior of all Mozilla users, independently of the specific version they use, while each original DTMC captures the behavior of users with a specific version of the Mozilla browser. Similarly, the DTMC synthesized for property (2) captures the probabilistic behavior of all mobile users independently of the specific type of device. The BEAR analysis engine evaluates the property for the synthesized DTMC using a model checker, PRISM in our experiments [23]. The approach does not depend on the model checker and works with other engines, such as MRMC [19], as well as with ad-hoc mechanisms for efficient runtime analyses, like the approach described by Filieri et al. [14, 15]. The BEAR analysis engine processed property (1) resulting in a probability of 0.006 that Mozilla users will contact a sale agent, and the property (2) resulting in 32.36 announcements displayed on average by mobile users.

5. BEAR MODELS AT WORK

BEAR captures information about user interactions with Web applications. In this section we illustrate how this information can be used to enrich or modify the initial requirements and to keep them current as user behaviors evolve over time. BEAR can thus be used to support the evolution of Web applications. Indeed, many Web applications are initially designed with little or no knowledge of how its final users will behave. Resorting to the experience collected for similar applications, if any, only provides rough data. In addition, user behaviors may change with their familiarity with the application and for many other reasons. The analysis of the DTMCs by means of probabilistic model checking produces information about navigation anomalies, emerging behaviours and new attitudes of users that well complements the initial incomplete requirements. We show how this can be done by illustrating the use of BEAR in different maintenance actions for the findyourhouse.com case study introduced in Section 3. The results discussed here refer to a log file composed of 400.000 entries that correspond to ten months of users' interactions. We ran the BEAR inference engine on the log file relying on the filters listed in the third column of Table 1, and we obtained 571 DTMCs. We first illustrate the use of BEAR for detecting navigation anomalies and then for inferring emerging behaviours and new attitudes of users.

5.1 Detecting Navigational Anomalies

A navigational anomaly is a difference between the actual and the expected user navigation actions. The expected navigation is what has been implemented in the application and is represented by the application's *site map*. The actual navigation is instead represented by a path on the DTMCs inferred by BEAR, which corresponds to actual navigations performed by users in reality. Navigational anomalies can be detected by comparing the DTMCs with the site map. By detecting navigation anomalies, we can find suggestions to improve the application. We can, for example, identify frequent users' workarounds that may witness the lack of some navigational features. As an example of navigational anomaly detection, let us compare the model produced by BEAR with the findyourhouse.com site map, by running the BEAR analysis engine with queries of this kind[6]:

$$\{\}\mathcal{P} =?[(X\, s_i)]\{s_j\}$$

for every state s_i, s_j in the model. This query specifies the request for the probability of a user to move from state s_j to state s_i. Because of the empty scope in the properties the analysis engine selected all the 571 inferred DTMCs and synthesized a unique model that represents the behaviour of the whole population of users. The synthesized model is composed of 23 states and 214 transitions and is partially reported in Figure 3. The BEAR analysis engine identified several navigational anomalies, despite the fact that the application has been in use for more than two years and has been carefully maintained and corrected. Here we illustrate one of them that corresponds to the two transitions exiting state s_7 of Figure 3 that occur with non-negligible probability in the DTMC, but do not correspond to transitions in the application site map. Thus, they represent frequent actions performed by users, which do not correspond to navigation features of the application. State s_7 corresponds to the *terms of use* page (proposition *tou*) that can be reached only from the *contacts* page (state s_5). The *terms of use* page does not offer a way to go back to the *contacts* page. The anomalous transitions that exit state s_7 correspond to the users trying to go back to the *contacts* page (state s_5), likely using the back button of the browser, and ending up in one of the states proceedings s_5 (states s_4 and s_6) because of the AJAX implementation of the application. The readers should notice that there is no transition from s_7 to s_1 that also precedes s_5. This indicates that the log file does not record any attempt of users to go back from s_7 to s_5 having reached s_5 directly from s_1. This is because s_1 correspond to the *home* page and no users felt the need to check the *terms of use* (s_7) before consulting renting or sales

[6]PRISM not only can evauate the truth or falsity of a property, but can also compute probability values.

announcements (states s_4 and s_6). This anomaly has been corrected by adding a back button to the *terms of use* page. Although navigational anomaly detection is presently done by manually inspecting and comparing the analysis results with a site map, the comparison can be easily automated by using available XML sitemap descriptions.

5.2 Inferring Behaviours and Attitudes

BEAR can analyze properties that correspond both to emerging behaviors and behaviors that may derive from new requirements. Application designers can use properties to describe the expected behaviors of either all or specific classes of users, as well as the impact of changes in the navigation attitude of users. Here we present the results of the analysis of the findyourhouse.com log file for properties that represent the different type of analyses. The findyourhouse.com owners were interested in improving the access to the application in terms of renting versus buying inquires. To do so, one needed to understand what is the probability of a user to browse sales and not renting announcements and, vice versa, the probability of users to browse renting announcements only. The former can be found with the query:

$$\{\}\mathcal{P} =?[(F\, sales\_anncs)\,\&\,(!(F\, renting\_anncs))]$$

The BEAR analysis engine indicated that 48% of users are interested in sales announcements only, and 20% users are interested in renting announcements only. The remaining percentage of users look both for sales and renting announcements. With these data, the designers decided to change the homepage that initially displayed a random set of announcements, and now displays renting and sales announcement proportionally to the measured users' behaviors. The findyourhouse.com owners were further interested in possible differences between mobile versus desktop users. They simply refined the above query by adding an ad-hoc scope to restrict the analysis to mobile users (see property (2) at page 7). The analysis engine selected 32 DTMCs out of the initial set of inferred DTMCs and synthesized a single model that captures the behavior of mobile users only, which indicated that 35% of the mobile users are interested in sales announcements only. With this additional information, the developers decided to properly customize the mobile home page. As a final example, the findyourhouse.com owners were planning a marketing campaign targeting desktop users with a banner published on social networks advertising the website and linked to the *sales_anncs* page. In this scenario it is crucial to accurately predict the number of database queries generated by the marketing campaign to suitably adapt the website infrastructure to prevent a potential denial of service. Similarly to the example illustrated in Section 4.4, developers associated the rewards that represent the number of data base queries needed to display each page of the application to the atomic propositions defined by filters to weight each state of the inferred DTMCs with a reward indicating the impact of that state in terms of queries to the database, and formulated the query:

$$\{(?!(.*)(Android \mid iOS))(.*)\}\mathcal{R}_{=?}[F\, end\, \{sales\_anncs\}]$$

The query, whose scope excludes Android and iOS users to just focus on desktop users, asks for the resource usage ($\mathcal{R}_{=?}$) after hitting the *sales_anncs* page. Since the resource usage is measured in terms of database queries, the analysis of the property gives the estimate of database queries of desktop users after accessing that page. The analysis engine selected 539 models (out of the initial 571 inferred models) that capture the behaviors of desktop user and synthesized an appropriate unique model against which to verify the property. The product between the obtained result and the expected click-through rate per hour (estimated from previous marketing campaigns or with ad-hoc techniques, like the one defined by Richardson et al. [29]) represents a reasonable prediction of the database queries generated by the marketing campaign. Similarly, we augmented the model with rewards that indicate the average amount of storage consumed by each page request to predict the storage requirements implied by the additional traffic of the marketing campaign. In general, rewards can model every resource that directly depends on users' actions and can be represented numerically to support capacity planning and forecasting analyses.

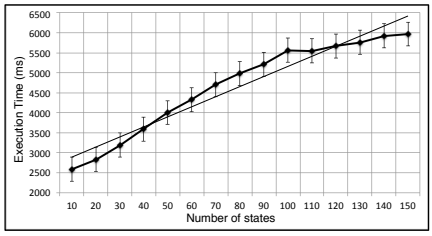
6. PERFORMANCE AND SCALABILITY

We evaluated the performance and scalability of both the BEAR inference and analysis engine on a 2 GHz Intel Core i7, 8GB RAM with Java™1.7.0. Each experiment has been repeated one hundred of times collecting the average result and the standard deviation.

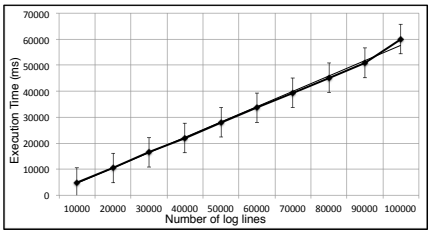
The execution time of the BEAR inference engine depends on two factors: (1) the number of states in the inferred model and (2) the length of the log file. We generated synthetic log files composed of $10,000$ lines generated ad-hoc to produce DTMCs with an increasing number of states. Figure 4(a) shows that the BEAR inference engine can produce a DTMC of 150 states analyzing a log files of $10,000$ lines in few seconds. Figure 4(b) illustrates instead the execution time of processing log files of increasing length. The figure reports the execution time for log files that produce a DTMC of 50 states. BEAR processes a log file of $100,000$ lines in around 60 seconds. inference engine.

The execution time of the BEAR analysis engine depends on two factors: (1) the number of synthesised models and (2) the number of states of synthesised models. In the experiments discussed below we shows the execution of the synthesis algorithm. We refer to [18] for the performance of evaluating PCTL properties on DTMCs. We randomly generated fully connected DTMCs composed by 50 states and we measured the execution time needed to synthesise a single model selecting an increasing number of input DTMCs. Figure 5(a) shows that the BEAR analysis engine can synthesise a DTMC from $1,000$ input DTMCs in few seconds. Figure 5(b) illustrates instead the execution time of synthesising models with an increasing number of states. The figure reports the execution time to synthesise 500 input DTMCs. Also in this case BEAR shows a reasonable execution time. The figure shows that models composed of 120 states are synthesised in around 25 seconds. It is important to notice that synthesised modes are not necessarily generated from scratch each time because of an internal caching mechanism. The log file of the case study is composed of over 400,000 entries, and we conducted all the experiments within few minutes, thus confirming the scalability of the approach. Our implementation has been released as an open source artifact[7] that includes an anonymized excerpt of the findyourhouse.com log file.

[7] `http://giordano.webfactional.com/?page_id=22`

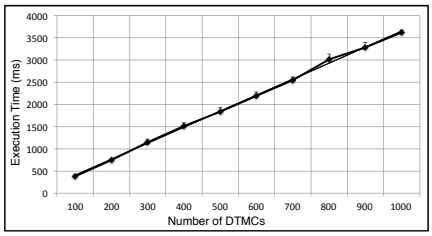


(a) Increasing number of states.

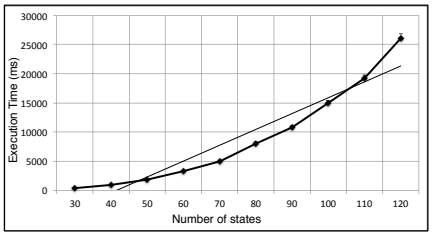


(b) Log of increasing length.

Figure 4: Inference engine performance evaluation



(a) Increasing number of DTMCs.



(b) Increasing number of states.

Figure 5: Analysis engine performance evaluation

7. RELATED WORK

Many existing works address the problem of model inference from execution traces tackling a wide range of research problems that range from mining API patterns [2], inferring specifications [25, 10, 30], inferring behavioral models for web-services and software processes [5, 9], testing Web applications [28, 34], and detecting faults [27, 26]. Worth to mention is also Perracotta [38], an approach to mine and visualize temporal properties of event traces used to study program evolution and the work by Krka et al. [21] that consists of an approach to mine invariants and improve precision of inferred models. Finally, it important to mention the work by Beschastnikh et al. [7, 33] that illustrates Synoptic, a tool that helps developers by inferring from log files a concise and accurate system model focusing on generating invariant-constrained models. Beschastnikh et al. also discuss in [6] an approach to specify inference algorithms declaratively. Complementary to these approaches there is the work by Tonella et al. [36] that discusses how to find the optimal approximation in the inference process. All these solutions represent fundamental tools for developing complex and dependable software systems even if, differently from BEAR, they do not focus on user behaviours that actually represent a crucial factor in applications domains such as user-intensive Web applications.

The problem of capturing and analysing the user behaviours in Web applications through the analysis of log files has been address by many approaches as reported in the survey of Facca et al. [12]. Many of these approaches focus on mining specific information to perform peculiar tasks. For example, Liu and V. Kešelj combine the analysis of Web server logs with the contents of the requested Web pages to predict users' future requests [24]. They capture the content of Web pages by extracting character N-grams that are combined with the data extracted from the log files. Schechter et al. in [32] use instead a tree-based data structure to represent the collection of paths inferred from the log file to predict the next page access. Similarly, Sarukkai [31] relies on Markov chains for link prediction and path analysis. Alternatively, Yang et al. [39] focus on predicting accesses for efficient data caching. These approaches, even if extremely helpful for the specific tasks they have been conceived for, lack of general applicability. Differently, the BEAR approach produces a navigational model that can be used to verify a plethora of different properties that span from simple navigational probabilities that may be used for link prediction to more complex properties that may be used for capacity planning analysis, as exemplified in the paper. Indeed, the flexibility of the proposed approach relies on the expressiveness of the PCTL formal logic that allows engineers to encode the properties to be verified including domain specific aspects of the application. Finally, it is worth to mention the work by Komuravelli et al [20] that illustrates the inference of a probabilistic system given finite set of positive and negative samples and the work by Chierichetti et al. [8] that discusses the approximation of Web users with Markov models.

8. CONCLUSIONS AND FUTURE WORK

We presented a novel inference mechanism conceived ad-hoc for probabilistic model checking to elicit requirements about emerging users' behaviours. The approach extracts DTMCs that represent the users' behaviours from application logs, and analyses them by means of probabilistic model checking to identify navigation anomalies and emerging users' behaviours. We are extending the approach with probabilistic timed automata [4] to capture other user behaviors.

9. REFERENCES

[1] Google Analytics. http://www.google.com/intl/en/analytics/.

[2] M. Acharya, T. Xie, J. Pei, and J. Xu. Mining api patterns as partial orders from source code: from usage scenarios to specifications. In *ESEC/FSE*, 2007.

[3] S. Andova, H. Hermanns, and J. Katoen. Discrete-time rewards model-checked. *Formal Modeling and Analysis of Timed Systems*, pages 88—104, 2004.

[4] C. Baier and J.-P. Katoen. *Principles of Model Checking*. The MIT Press, 2008.

[5] A. Bertolino, P. Inverardi, P. Pelliccione, and M. Tivoli. Automatic synthesis of behavior protocols for composable web-services. In *ESEC/FSE*, 2009.

[6] I. Beschastnikh, Y. Brun, J. Abrahamson, M. D. Ernst, and A. Krishnamurthy. Unifying fsm-inference algorithms through declarative specification. In *ICSE*, pages 252–261. IEEE Press, 2013.

[7] I. Beschastnikh, Y. Brun, S. Schneider, M. Sloan, and M. Ernst. Leveraging existing instrumentation to automatically infer invariant-constrained models. In *ESEC/FSE*, 2011.

[8] F. Chierichetti, R. Kumar, P. Raghavan, and T. Sarlós. Are web users really markovian? In *WWW*, pages 609–618. ACM, 2012.

[9] J. E. Cook and A. L. Wolf. Discovering models of software processes from event-based data. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 7(3):215–249, 1998.

[10] G. de Caso, V. Braberman, D. Garbervetsky, and S. Uchitel. Program abstractions for behaviour validation. In *ICSE*. IEEE, 2011.

[11] M. DeGroot and M. Schervish. *Probability and Statistics-International Edition*. Addison-Wesley. Publishing. Company., Reading, Massachusetts, 2001.

[12] F. Facca and P. Lanzi. Mining interesting knowledge from weblogs: a survey. *Data & Knowledge Engineering*, 53(3):225–241, 2005.

[13] R. Fielding and R. Taylor. Principled design of the modern web architecture. *ACM Transactions on Internet Technology (TOIT)*, 2(2):115–150, 2002.

[14] A. Filieri, C. Ghezzi, and G. Tamburrelli. Run-time efficient probabilistic model checking. In *ICSE*, 2011.

[15] A. Filieri and G. Tamburrelli. Probabilistic verification at runtime for self-adaptive systems. In *Assurances for Self-Adaptive Systems*, pages 30–59. Springer, 2013.

[16] L. Grunske. Specification patterns for probabilistic quality properties. In *ICSE*, pages 31–40. IEEE, 2008.

[17] H. Hansson and B. Jonsson. A logic for reasoning about time and reliability. *Formal aspects of computing*, 6(5):512—535, 1994.

[18] D. N. Jansen, J.-P. Katoen, M. Oldenkamp, M. Stoelinga, and I. Zapreev. How fast and fat is your probabilistic model checker? an experimental performance comparison. In *Hardware and Software: Verification and Testing*, pages 69–85. Springer, 2008.

[19] J. Katoen, M. Khattri, and I. Zapreevt. A markov reward model checker. In *Quantitative Evaluation of Systems, 2005. Second International Conference on the*, pages 243–244. IEEE, 2005.

[20] A. Komuravelli, C. S. Pasareanu, and E. M. Clarke. Learning probabilistic systems from tree samples. In *LICS*, pages 441–450. IEEE, 2012.

[21] I. Krka, Y. Brun, D. Popescu, J. Garcia, and N. Medvidovic. Using dynamic execution traces and program invariants to enhance behavioral model inference. In *ICSE*, volume 2. IEEE, 2010.

[22] M. Kwiatkowska, G. Norman, and D. Parker. Stochastic model checking. In M. Bernardo and J. Hillston, editors, *Formal Methods for Performance Evaluation*, volume 4486 of *Lecture Notes in Computer Science*, pages 220—270. Springer, 2007.

[23] M. Kwiatkowska, G. Norman, and D. Parker. PRISM 4.0: Verification of probabilistic real-time systems. In G. Gopalakrishnan and S. Qadeer, editors, *CAV 2011*, volume 6806 of *LNCS*, pages 585–591. Springer, 2011.

[24] H. Liu and V. Kešelj. Combined mining of web server logs and web contents for classifying user navigation patterns and predicting users‘ future requests. *Data & Knowledge Engineering*, 61(2):304–330, 2007.

[25] D. Lo and S.-C. Khoo. Smartic: towards building an accurate, robust and scalable specification miner. In *FSE*, pages 265–275. ACM, 2006.

[26] D. Lo, L. Mariani, and M. Pezzè. Automatic steering of behavioral model inference. In *FSE*. ACM, 2009.

[27] L. Mariani, F. Pastore, and M. Pezzè. Dynamic analysis for diagnosing integration faults. *Software Engineering, IEEE Transactions on*, 37(4), 2011.

[28] A. Mesbah and A. Van Deursen. Invariant-based automatic testing of ajax user interfaces. In *ICSE*. IEEE, 2009.

[29] M. Richardson, E. Dominowska, and R. Ragno. Predicting clicks: estimating the click-through rate for new ads. In *WWW*. ACM, 2007.

[30] M. Robillard, E. Bodden, D. Kawrykow, M. Mezini, and T. Ratchford. Automated api property inference techniques. *TSE*, 2012.

[31] R. Sarukkai. Link prediction and path analysis using markov chains. *Computer Networks*, 33(1), 2000.

[32] S. Schechter, M. Krishnan, and M. Smith. Using path profiles to predict http requests. *Computer Networks and ISDN Systems*, 30(1):457–467, 1998.

[33] S. Schneider, I. Beschastnikh, S. Chernyak, M. D. Ernst, and Y. Brun. Synoptic: summarizing system logs with refinement. *Proc. of SLAML*, 2010.

[34] M. Schur, A. Roth, and A. Zeller. Mining behavior models from enterprise web applications. In *FSE 2013*.

[35] J. Srivastava, R. Cooley, M. Deshpande, and P.-N. Tan. Web usage mining: discovery and applications of usage patterns from web data. *SIGKDD Explor. Newsl.*, 1(2):12–23, Jan. 2000.

[36] P. Tonella, A. Marchetto, C. D. Nguyen, Y. Jia, K. Lakhotia, and M. Harman. Finding the optimal balance between over and under approximation of models inferred from execution logs. In *ICST*, 2012.

[37] E. Wilde and C. Pautasso. *REST: From Research to Practice*. Springer, 2011.

[38] J. Yang, D. Evans, D. Bhardwaj, T. Bhat, and M. Das. Perracotta: mining temporal api rules from imperfect traces. In *ICSE*. ACM, 2006.

[39] Q. Yang and H. H. Zhang. Web-log mining for predictive web caching. *Knowledge and Data Engineering, IEEE Transactions on*, 15(4), 2003.