A Model for Usage-based Testing of Event-driven Software

Steffen Herbold, Jens Grabowski, Stephan Waack
Georg-August-Universität Göttingen
Intitute of Computer Science
Göttingen, Germany
Email: {herbold, grabowski, waack}@cs.uni-goettingen.de

***Abstract*—Event-driven software is very diverse, e.g., in form of *Graphical User Interface*s (GUIs), Web applications, or embedded software. Regardless of the application, the challenges for testing event-driven software are similar. Most event-driven systems allow a huge number of possible event sequences, which makes exhaustive testing infeasible. As a possible solution, *usage-based testing* has been proposed for several types of event-driven software. However, previous work has always focused on one type of event-driven software. In this paper, we propose a usage-based testing model for event-driven software in general. The model is divided into three layers to provide a maximum of platform independence while allowing interoperability with existing platform dependent solutions.**

I. Introduction

Event-driven software plays an important role in todays software systems. For example, most end-user software has a *Graphical User Interface* (GUI) that communicates with the user via events, e.g., mouse events. Another example is the *Service Oriented Architecture* (SOA), where service providers and service consumers communicate with each other using events. Other types of event-driven software include Web applications in general, network protocols, and embedded software. These software types are very diverse and are, therefore, seldom considered together when it comes to quality assurance. From a quality assurance point of view, these systems are quite similar. Therefore, event-driven testing methodologies can be applied to all of them.

One obstacle for the quality assurance of event-driven software is that the event space is often huge and the number of possible event sequences even larger, which makes exhaustive testing infeasible. The number of event sequences is often exponential in the number of possible events. One approach to resolve this scalability problem is *usage-based testing*, a goal oriented testing methodology that prioritizes testing effort based on how often which functionality of the system is used. The system's usage is described by *probabilistic usage profiles*, e.g., *Markov chains* that describe the probability of the next event. To obtain these profiles, *usage mining* is used, i.e., analyzing logged executions of the system's usage. Usage-based testing and usage mining for event-driven software are no new ideas, since a lot of research has already been performed, e.g., on web usage mining [1], [2], [3], usage-based testing of web applications [4], [5], [6], and usage-based based GUI testing [7], [8], [9].

The contribution of this paper is a usage-based testing model for event-driven software in general, i.e., independent of the type of event-driven software. To achieve a maximum of platform independence, but also interoperatbility with existing platform dependent solutions, the model is divided into three layers: the platform layer, the translation layer, and the event layer.

The platform layer represents the parts of the model, that depend on the platform on which the *System Under Test* (SUT) is implemented, i.e., the GUI, Web service, or other event-driven platform of the system. The platform itself has *platform specific* event types and targets, e.g., “mouse clicks on GUI objects” or “file transfer request to an *Internet Protocol* (IP) adress”. The translation layer translates the platform specific events into *abstract events* that only consist of string representation of both the event type and event target. Furthermore, the platform specific events may be to fine-grained for usage modeling, e.g., events for both pressing the mouse button and releasing it instead of one event for the click. The translation layer allows the integration of already existing platform dependent solutions into our model. The event layer is platform independent. It works on abstract events and provides functionality for the training of probabilistic usage profiles, analysis of these profiles, and testing task, like test case generation.

In addition to the testing model, we summarize the most important stochastic models used for the definition of probabilisitic usage profiles. The usage profiles are at the center of the model and therefore require special attentation. We compare these models and describe their strengths and weaknesses.

The remainder of this paper is structured as follows. In Section II, we define the structure of the testing model. Afterwards, the layers of the models are discussed in detail. In Section III the platform layer and its main components are described. In Section IV, the translation layer and its task as a mediator between the other two layers are described. The event layer is defined in Section V. In Section VI, stochastic models that are commonly used to define probabilistic usage profiles are described and compared . In Section VII, we outline the future research of this work. Finally, the paper is concluded in Section VIII.

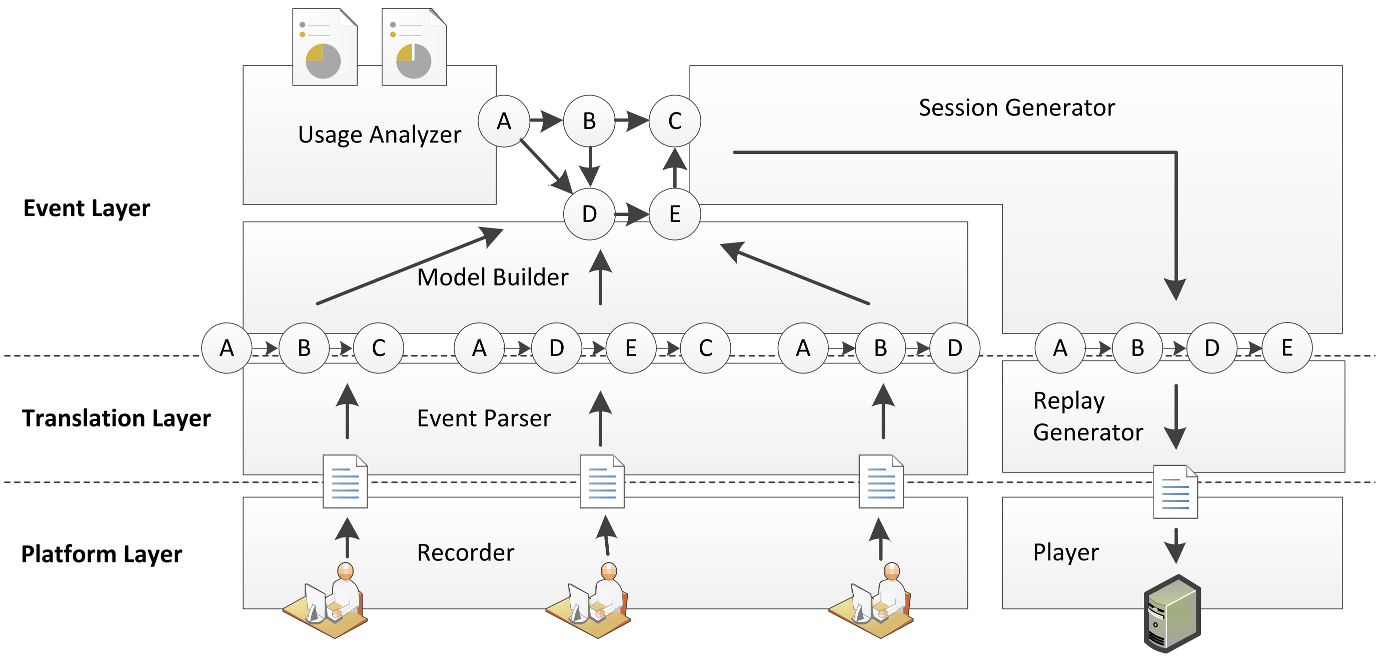


Fig. 1. Model Overview

II. Model Outline

The testing model for event-driven software (Figure 1) we propose has three layers: the platform layer, the translation layer, and the event layer. In the following, the tasks and responsibilities of each layer are presented.

The lowest layer is the platform layer. It contains the platform specific parts of the model, i.e., a *recorder* to monitor the SUT and log its execution and a *player* that can send stimuli to the SUT to replay event patterns. Above the platform layer is the translation layer, whose main task is to mediate between the platform layer and the platform independent event layer. To accomplish this, the translation layer provides an *event parser* to convert the logs produced by the recorder into event traces and a *replay regenerator* to generate input for the player from event traces. The event layer is platform independent and works on the abstract concept of events.

Definition 1. An *event* e is an observable action of a system that is characterized by its *type* and its *target*, i.e., $e = \{type(e), target(e)\}$.

The possible event types depend on the platform and the targets depend on the SUT. While the events themselves are therefore platform dependent, the abstract notion of events that consist of a type and a target is platform independent. An example for a GUI event is $\{type : LeftMouseClick, target : Button.OK\}$ for a click on an OK button. The event layer provides a *model builder* that generates probabilistic usage models from event traces. The *usage analyzer* examines these models and provides statistical information about the SUTs usage. A *session generator* generates new event sequences from the usage models. These sequences can be used as test cases.

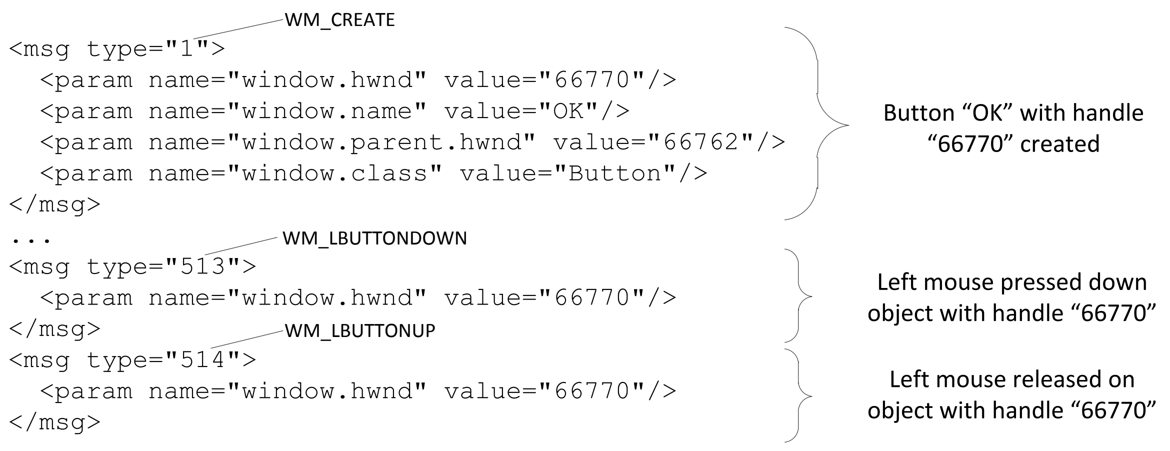
Figure 1 shows the interaction between the layers and their main components. The logs produced by the recorder are translated into event sequences by the event parser. These sequences are then utilized by the event layer components. The session generator of the event layer produces event sequences that are translated by the replay generator into a format that can be used by the player. The player can then execute the generated sessions on the SUT.

III. Platform Layer

The platform layer is the lowest of the three layers and responsible for the platform specific parts of the model. A platform is *"a set of subsystems and technologies that provide a coherent set of functionality through interfaces and specified usage patterns, which any application supported by that platform can use without concern for the details of how the functionality provided by the platform is implemented"* [10]. Some examples for platforms are the *Java EE Platform*[1] for network applications, the *Eclipse Rich Client Platform* (RCP)[2], and the *Microsoft Foundation Classes* (MFC)[3] framework for Windows application development. The two main components of the platform layer are the recorder and the player.

The recorder monitors and logs the usage of the SUT. For example, in the context of GUI usage, this means the monitoring of mouse clicks and keyboard input. For Web applications, the communication between service providers and costumers needs to be monitored. The recorder can either be internal, i.e., integrated into the SUT or external, i.e., as a seperate tool. The advantage of internal recorders is that they can easily be

[1]http://www.oracle.com/technetwork/java/javaee/overview/index.html
[2]http://www.eclipse.org/home/categories/rcp.php
[3]http://msdn.microsoft.com/en-us/library/d06h2x6e.aspx



```
<msg type="1">
  <param name="window.hwnd" value="66770"/>
  <param name="window.name" value="OK"/>
  <param name="window.parent.hwnd" value="66762"/>
  <param name="window.class" value="Button"/>
</msg>
...
<msg type="513">
  <param name="window.hwnd" value="66770"/>
</msg>
<msg type="514">
  <param name="window.hwnd" value="66770"/>
</msg>
```

WM_CREATE
Button “OK” with handle “66770” created
WM_LBUTTONDOWN
Left mouse pressed down object with handle “66770”
WM_LBUTTONUP
Left mouse released on object with handle “66770”

Fig. 2. Examplary log exerpt produces by a GUI recorder

shipped to costumers as part of the software, whereas external recorders require seperate deployment, may need different licencing, or cannot be shipped at all. In a previous work, we presented an internal recorder for the monitoring of Windows applications built on top of MFC framework [11]. The recorder produces a log that contains platform specific events. In case of using Windows as a platform, these events are *messages*, e.g., `WM_LBUTTONDOWN` and `WM_LBUTTONUP` for pressing the left mouse button down and releasing it. The platform-specific events are not the same as the events used for usage modelling. Often, the usage events are defined by several platform specific events. For example, in MFC applications, usage events of type *LeftMouseClick* consist of the two platform specific events `WM_LBUTTONDOWN` and `WM_LBUTTONUP`.

The player is the counterpart of the recorder. It has to execute the SUT by sending stimuli. On most platforms, the stimuli are closely related to platform specific events monitored by the recorder. In case of GUI platforms, these stimuli are mouse and keyboard inputs. We suggest such a player for the Windows MFC platform in a previous work [11]. The concept of a recorder and a player for the monitoring and replaying of executions is widely adapted for the testing of event-driven software by *capture/replay* tools. The recorders of these applications are always external. Implementations are available for many platforms, e.g., jRapture for Java [12], or IBM Rational Robot[4] which supports many platforms, including Java, .NET, and Web applications.

IV. Translation Layer

The translation layer provides a bridge between the platform layer and the event layer, and bundles all interdependencies between the two layers, which has many advantages. Foremost, it completly decouples the platform layer from the event layer. Given existing platform and event layer implementations, only an appropriate translation layer implementation is required for the two to work together. Neither the platform, nor the event layer themselves need to be adapted. This is of particular importance, as it provides interoperability with already existing solutions, e.g., capture/replay tools.

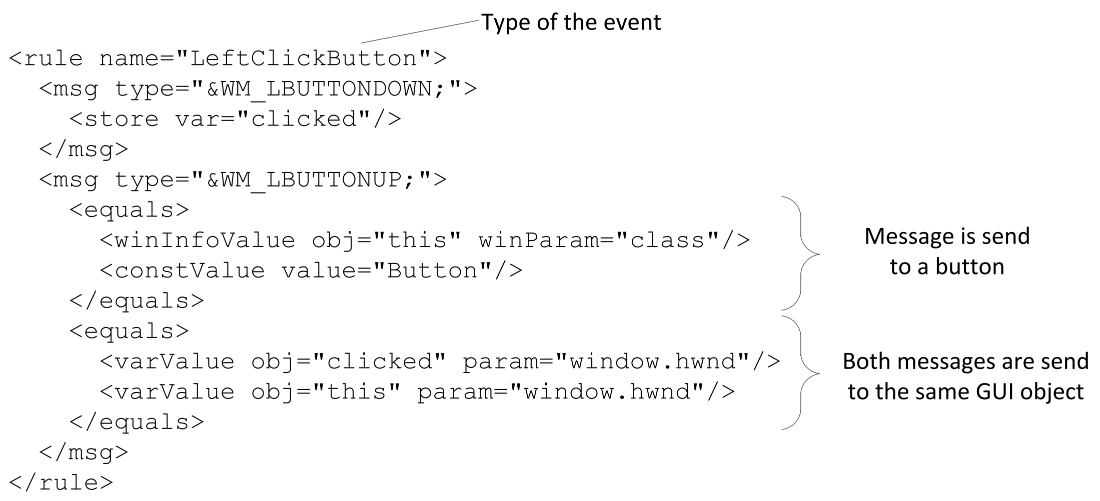
The platform specific events are often not the events that are important for usage modelling. Figure 2 shows a short log excerpt generated by an internal recorder for Windows MFC applications [11]. The first message signifies that an “OK” button with handle 66770 has been created. The handle can be considered as unique ID of the button and can therefore be used to identify it. The following two messages signify that the left mouse button has been pressed down and released upon a GUI object with handle 66770, i.e., the “OK” button. For usage modeling, these messages separately hold no interest, as none of them describe an event by themselves. However, in combination they describe the event $\{type : LeftClickButton, target : Button.OK\}$.

The first task of the translation layer is to convert logs that are monitored by recorders and convert platform specific events into abstract usage events. For this purpose, the translation layer provides an *event parser*. In case the platform specific events are the same as the abstract usage events, the event parser does nothing and simply forwards the events as they are. However, in cases as exemplified in Figure 2, the events have to be infered from the log. A set of rules that describes the structure of events is required for this task. An example for such a rule defined in XML is shown in Figure 3. This rule can be applied to the messages in Figure 2 and it means that an event of type *LeftClickButton* consists of the two messages `WM_LBUTTONUP` and `WM_LBUTTONDOWN` on a GUI object that is a “Button”. As a result, the event parser produces *sessions* of events.

Definition 2. A *session* s is an ordered sequence of events $s = (e_1, e_2, \ldots, e_m)$ with $e_1, \ldots, e_m \in E$. The set S_U contains all *user sessions*, i.e., all sessions s observed for the users U.

The second task of the translation layer is to convert the abstract events back into a representation that can be used by platform level players. This can be thought of as the inverse operation of the event parser and it is performed by a *replay generator*.

[4]http://www-01.ibm.com/software/awdtools/tester/robot/



```
<rule name="LeftClickButton">
  <msg type="&WM_LBUTTONDOWN;">
    <store var="clicked"/>
  </msg>
  <msg type="&WM_LBUTTONUP;">
    <equals>
      <winInfoValue obj="this" winParam="class"/>
      <constValue value="Button"/>
    </equals>
    <equals>
      <varValue obj="clicked" param="window.hwnd"/>
      <varValue obj="this" param="window.hwnd"/>
    </equals>
  </msg>
</rule>
```

Type of the event
Message is send to a button
Both messages are send to the same GUI object

Fig. 3. An event parser rule

V. Event Layer

The event layer is responsible for the training of usage profiles and their exploitation. This includes various tasks from statistical analysis to test case generation. The event layer is built on top of the abstract notion of events (Definition 1). The possible events of the system are defining the event space.

Definition 3. The *event space E* is the set of all theoretically possible events e of a system. The *observed event space $E_U \subseteq E$* is the set of all events, that were observed during sessions S_U of the users U.

The size of the event space is an important property both for testing and model analysis. It can be used as an estimator for the required test suite size or as a normalization factor for analysis results. However, the size of the event space E has to be determined externally. It cannot be determined by simply logging user actions, as there is no guarantee that the users utilize all functionality offered by the system. The observed event space E_U is a lower bound for the size of E and can be used to estimate its size. The quality of this estimation depends on the versatility the software and the number of its functions that are actually used. The estimation of the event space size is out of the scope of this work. A way to infer the event space for GUI applications is outlined in [13]. We assume that the event space E is available or that E_U is a sufficiently good estimator.

In the following, we define probabilisitic usage profiles and outline how they are trained based on the event sessions provided by the event parser. Furthermore, we present a method for the extraction of statistical information about the usage of the software from the usage profiles and show how these information can be utilized. Afterwards, we show how these profiles can be exploited to randomly generate sessions that can be used as basis for new test cases.

A. Probabilistic Usage Profiles

Usage profiles are a means for the modelling of how a system is used by its users. In the literature, the terms usage profile and *user profile* are sometimes mixed up. A user profile does not describe how the software is used, but rather attributes of the users themselves, e.g., the age or the gender. There are two types of usage profiles: those that take the internal state of the system into account and those that only consider the events themselves. In this work, we refer to the second type, i.e., the underlying models make no assumptions about the internal state of the system.

In case the behavior of the users is modelled using probabilities, i.e., "after event e, the next event will be e' with probability 0.2", we speak of *probabilistic usage profiles*. These types of profiles are *discrete stochastic processes* over the event space.

Definition 4. A *discrete stochastic process* is an indexed sequence of random variables $X_1, X_2, \ldots$ and is characterized by the joint probability mass function $\Pr\{(X_1, X_2, \ldots, X_n) = (x_1, x_2, \ldots, x_n)\} = p(x_1, x_2, \ldots, x_n)$ with $(x_1, x_2, \ldots, x_n) \in \mathcal{H}^n$ for $n \in \mathbb{N}$.

Definition 5. A *probabilistic usage profile* is a discrete stochastic process over the event space as alphabet $\mathcal{H} = E$.

To implement the event layer, a concrete type of stochastic process has to be chosen. In Section VI, some of the often used types are described and compared. The usage profiles themselves are infered by the *model builder*. How this training works, depends solely on the underlying stochastic process used. The training data are the observed user sessions S_U.

B. Usage Analysis

There are several ways to analyze the usage of a system based on user sessions and probabilistic usage profiles. We will now outline how usage data can be used to answer the question which features of a system are used most often. This analysis is important for almost everyone in a project, as development, testing, and marketing should focus on these popular features.

One way to estimate how often an event occurs is to obtain its *empirical likelyhood* based on the logged usage data. The empirical likelyhood of an event $e \in E$ given the usage data S_U is $\frac{\sum_{s \in S_U} count(e,s)}{\sum_{s \in S_U} |s|}$, where $count(e,s)$ is number of occurences of the event e in the session s. Another way to obtain this information is to analyze the stochastic process underlying the probabilistic usage profile. To obtain a measure for the probability of an event based on the process, the *stationary distribution* can be used.

Definition 6. A stochastic process $X_1, X_2, \ldots$ is called *stationary* if its joint distribution is invariant to timeshifts, i.e.,

$$\begin{aligned} &\Pr\{X_1 = x_1, X_2 = x_2, \ldots, X_n = x_n\} \\ &\quad = \Pr\{X_{1+l} = x_l, X_{2+l} = x_2, \ldots, X_{n+l} = X_n\}. \end{aligned} \tag{1}$$

Stationary processes have a *stationary distribution* μ, such that $\mu(x) = \lim_{t \to \infty} p(X_t = x)$ for all $x \in \mathcal{H}$.

The stationary distribution describes the asymptotical probability that an event occurs. Therefore, the stationary distribution describes for each event e the probability $\mu(e)$ of its occurence in the process, i.e., the probability that a user triggers the event. Not every stochastic process is stationary and it has to be guaranteed that the process is indeed stationary before this method can be applied.

The raw probabilites themselves are hard to interpret, as they tend to be very small, especially with large state spaces. To mitigate this, the state space size $|E|$ has to be used as a normalization factor to identify how often features are used in relation to the number of features in total. In general, $|E|$ is an important normalization factor for usage analysis.

These are only two examples for the analysis of usage. Further tasks may include the estimation of the usage *entropy* [14], i.e., how random the usage of the system is. Another way to exploit analysis results is to prioritize test cases based on the obtained information. Additionally, all analysis can also focus on the event types or targets. For example, has the user rather used the mouse or the keyboard, or which buttons were clicked most often.

C. Randomized Session Generation

An important way to exploit probailistic usage profiles is *randomized session generation*, i.e., the generation of event sequences from the profile. This can be done by *randomly walking* the through the stochastic process. The random walk is the process of succesively drawing events one after another according to the probabilitiy distribution of the stochastic process. The result is then a randomly drawn user session. Formally, let $s_1^n = (e_1, \ldots, e_n)$ the randomly drawn session so far. Note, that n is zero and the session empty at the start of the procedure. The next event e is then drawn according to the probability distribution $\Pr\{e|s_1^n\}$.[5] The generated sessions can then be used as a basis for new test cases by adding appropriate *assertions*, i.e., checks values and states of the SUT are as expected. The current model is not able to do this automatically. To be able to generate test cases automatically, the usage profile needs to be enriched with information about the systems state.

[5]The notation $\Pr(e|s_1^n)$ is an abbreviation for $\Pr(X_{n+1} = e|X_n = e_n, \ldots, X_1 = e_1)$.

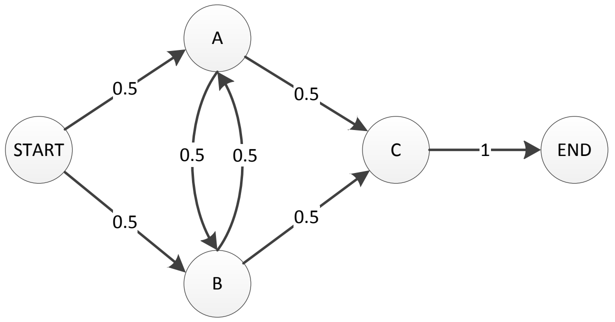


Fig. 4. A simple first-order Markov model

VI. Stochastic models

In this section, we summarize the most important stochastic models used to define probabilisitc usage profiles. All of the stochastic models introduced are of the same type, i.e., they have a *memory* of the previous events.

A. First-order Markov Models

The most popular choice for usage profiles are first-order *Markov Model*s (MMs), also known as *Markov chains* used by, e.g., [6], [7]. MMs are stochastic processes that fulfill the *Markov property*.

Definition 7. A discrete stochastic process $X_1, X_2, \ldots$ is said to possess the *Markov property* if $\Pr\{X_n = x_n | X_{n-1} = x_{n-1}, \ldots, X_1 = x_1\} = \Pr(X_n = x_n | X_{n-1} = x_{n-1}\}$ for all $n \in \mathbb{N}$.

In other words, such models are memoryless. Only the last event influences the probability of the next event. The size of the models is in $O(|E|^2)$, as for all events $e \in E$ the probability of the next event $p(e'|e)$ for all events $e' \in E$ has to be known.

The drawback of first-order MMs is that there is no guarantee that all sequences that can be generated by randomly walking through the MM are valid, as the internal state of the SUT is not taken into account. Consider the case with three events A, B, and C, where both A and B are precursors for C, however, the order of A and B is not specified. Then $s_1 = (A, B, C)$ and $s_2 = (B, A, C)$ are valid sessions, however, $s_{invalid} = (A, C)$ not. A first-order MM trained using s_1 and s_2 is depicted in Figure 4. In addition the events A, B, and C the model depicted in Figure 4 has global START and END states. The existence of the START state is based on the assumption that the SUT is in the same state at the beginning of each user session. The END state is implicitly added to the end of all user sessions and can be used as a means to determine the end of a random walk.

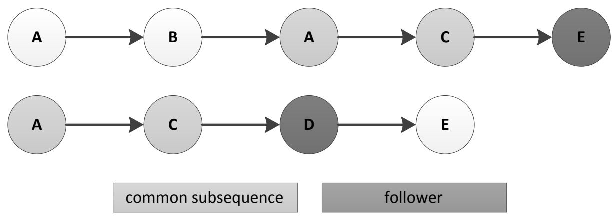


Fig. 5. Common Subsequences

Because the model is memoryless, it is not possible to know whether B already took place, when A was the last event. Therefore, there is a 50% chance that the next event is C, as this was the case in 50% of the cases in the training data. Therefore, there is a possiblity that $s_{invalid}$ is the result of a random walk through the model.

B. High-order Markov Models

A remedy for this problem is to use models with a memory, i.e., where the next event does not solely depend on the previous event but on the previous k events. To allow such a memory, the Markov property can be generalized as follows.

Definition 8. A discrete stochastic process $X_1, X_2, \ldots$ is said to possess the the *k-th order Markov property* if $\Pr\{X_n = x_n | X_{n-1} = x_{n-1}, \ldots, X_1 = x_1\} = \Pr(X_n = x_n | X_{n-1} = x_{n-1}, \ldots, X_{n-k} = x_{n-k}\}$ for all $n \in \mathbb{N}$.

Note, that this definition is equivalent to Definition 7 for $k = 1$. A stochastic process is called a *k-th order MM* if it fulfills the k-th order Markov property. Examples for research based on high order MMs are [15], [16]. The size of a k-th order MM is in $O(|E|^{k+1})$, as for each combination of events $e_1, \ldots, e_k \in E$ the probability $p(e'|e_k, e_{k-1}, \ldots, e_1)$ has to be known.

While using long memories helps to prevent the generation of invalid sessions when randomly walking through the model, there are some major drawbacks. The first is that the size grows exponentially with k. Even under assumption that many probabilities are zero and the model can be stored sparsely[6], the size eventually gets out of hand with larger event spaces and longer memories. Therefore, k has to be choosen reasonably low, as the model does not scale otherwise.

The second drawback of long histories is that the models become static instead of probabilistic. Consider the extreme case, where the length of the memory is the same as the length of the training sequences. In this case, the model is simply a collection of all training sequences, without any generalization at all. This phenomenom, where the model is specifically tailored to the training data is called *overfitting*. While this may be an extreme case, it shows the danger of long histories. In order to produce randomness in the model, the memory must not be longer than the length of the longest common subsequence of the training data. Consider the two sequences shown in Figure 5. The longest common subsequence is A, C. The following symbol is different in both sequences, E in the first one and D in the second one. Therefore, given a memory length of $k = 2$, there is a random element in the model, i.e., "what comes after A, C". For $k \geq 3$, the model simply learns the two sequences and degenerate into a non-probabilistic collection of sequences.

In conclusion, high-order MM with a fixed memory length can solve the problems if there are several preconditions without a fixed order for an event to occur. However, using long memories do not scale well and may remove the randomness from the model so that it degenerates into a collection of sequences.

C. Variable-order Markov Models

The first-order and high-order MMs both have drawbacks due to their fixed history length. A possible solution to this problem is to use *Variable-order Markov Model*s (VMMs). VMMs also fulfill the k-th order Markov property, but k is not fixed, i.e., at different times, different memory lengths are allowed. VMMs used in research are, e.g., *all k-th order Markov Models* [17], [18] and *Prediction by Partial Match*s (PPMs) [19], [20]. These models combine the ability to include long-term memories with the advantages of the increased randomness of models with only short-term memories. The size of the models varies, however, it is asymptotically usually not worse than high-order MMs.

To exemplify how VMMs introduce further randomness into long-memory models, we outline the PPM [21], [22] approach. In principle, PPM can be considered as a k-th order MM, that is extended by an *escape* mechanism. The escape mechanism offers the model the possibility to "opt-out" of the underlying k-th order MM and instead use shorter memories. This is done by allocating a probability $\hat{\mathbb{P}}_k(escape|s)$ for all events e that were not observed after the (sub)sequence s. The remainder of the probability mass is distributed among the observed symbols, according to the training of the model. Due to the non-zero proability of unobserved events, the escape mechanism allows PPM models to generate previously unobserved event combinations. The probability of the next event being $e \in E$ given that the current memory is the subsequence $s_{n-k}^n = (e_{n-k}, \ldots, e_n)$ is recursively defined as

$$\Pr_k(e|s_{n-k}^n) = \begin{cases} \hat{\Pr}_k(e|s_{n-k}^n) & \text{if } s_{n-k}^n \in S_U \\ \hat{\Pr}_k(escape|s_{n-k}^n) \cdot \Pr_{k-1}(e|s_{n-(k-1)}^n) & \text{otherwise} \end{cases} \tag{2}$$

with S_U being the user sessions used to train the PPM. How $\hat{\Pr}_k$ is calculated depends on the PPM variant used, as there are several implementations of the escape mechanism.

The strength of the escape mechanism is also its drawback, as it allows the generation of event combinations that are not part of the training data and were, therefore, not observed. This can lead to invalid generated sequences. Therefore, the escape mechanism needs to be implemented in a way that minimizes the possiblity of generating invalid traces. However,

[6]Only non-zero values are stored. In case nothing is stored for a given combination, the value is zero.

given an appropriate escape implementation, PPM has all the advantages of high-order MMs without one major drawback, i.e., their inflexibility.

VII. Future Work

The future work on this topic will focus in two directions. On one hand, we plan to further improve the testing model presented in this paper. To this aim, we will investigate how further testing tasks can be integrated into the event layer, e.g., test case prioritization. Furthermore, we will try to adapt the model in such a way that it is possible to integrate information about the SUT's state. This is an important step towards the automatic generation of not only sessions, but whole test cases.

On the other hand, we will work on the implementation of the model. This includes platform layer implementations for one or more platforms, according translation layer implementations and the implementation of the event layer features depicted. Furthermore, we will investigate the effect of the stochastic model used for the probabilistic usage profiles, especially different VMM variants. Using this implementation, case studies will be conducted to validate the feasability of the approach and the applicability of the testing approach to real-world software systems.

VIII. Conclusion

In this paper, we presented a model for usage-based testing of event-driven software systems. The model is designed to be independent of a specific platform. For this purpose, three layers are used to seperate the platform dependent parts of the model from the platform independent ones. The platform layer contains the platform-specific implementations for monitoring and replaying of events, the translation layer provides functionality to describe the collected data in an abstract and platform independent way. The abstract data is then used in the event layer for various tasks, e.g., the training of probabilistic usage profiles, the analysis of the SUT's usage, or usage-based session generation. Furthermore, we have presented the most important stochastic models used to define probabilistic usage profiles and compared their strengths and weaknesses. In future work, we will further extend and implement the approach, as we believe it has much potential.

References

[1] B. Mobasher, H. Dai, T. Luo, and M. Nakagawa, "Discovery and Evaluation of Aggregate Usage Profiles for Web Personalization," *Data Min. Knowl. Discov.*, vol. 6, no. 1, pp. 61–82, 2002.

[2] R. Cooley, "The use of web structure and content to identify subjectively interesting web usage patterns," *ACM Trans. Internet Technol.*, vol. 3, no. 2, pp. 93–116, 2003.

[3] M. Eirinaki and M. Vazirgiannis, "Web mining for web personalization," *ACM Trans. Internet Technol.*, vol. 3, no. 1, pp. 1–27, 2003.

[4] S. Schechter, M. Krishnan, and M. D. Smith, "Using path profiles to predict HTTP requests," *Comput. Netw. ISDN Syst.*, vol. 30, no. 1-7, pp. 457–467, 1998.

[5] C. Kallepalli and J. Tian, "Measuring and Modeling Usage and Reliability for Statistical Web Testing," *IEEE Trans. Softw. Eng.*, vol. 27, no. 11, pp. 1023–1036, 2001.

[6] P. Tonella and F. Ricca, "Statistical testing of web applications," *J. Softw. Maint. Evol.*, vol. 16, no. 1-2, pp. 103–127, 2004.

[7] J. A. Whittaker and M. G. Thomason, "A Markov Chain Model for Statistical Software Testing," *IEEE Trans. Softw. Eng.*, vol. 20, no. 10, pp. 812–824, 1994.

[8] I. Alsmadi, "The Utilization of User Sessions in Testing," in *ICIS '08: Proceedings of the Seventh IEEE/ACIS International Conference on Computer and Information Science (icis 2008).* Washington, DC, USA: IEEE Computer Society, 2008, pp. 581–585.

[9] P. A. Brooks and A. M. Memon, "Automated gui testing guided by usage profiles," in *ASE '07: Proceedings of the twenty-second IEEE/ACM international conference on Automated software engineering*. New York, NY, USA: ACM, 2007, pp. 333–342.

[10] A. G. Kleppe, J. Warmer, and W. Bast, *MDA Explained: The Model Driven Architecture: Practice and Promise*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 2003.

[11] S. Herbold, U. Bünting, J. Grabowski, and S. Waack, "Improved Bug Reporting and Reproduction trough Non-intrusive GUI Usage Monitoring and Automated Replaying," in *Third International Workshop on Testing Techniques & Experimentation Benchmarks for Event-Driven Software (TESTBEDS 2011).* IEEE Computer Society, 2011.

[12] J. Steven, P. Chandra, B. Fleck, and A. Podgurski, "jRapture: A Capture/Replay tool for observation-based testing," *SIGSOFT Software Engineering Notes*, vol. 25, no. 5, pp. 158–167, 2000.

[13] A. M. Memon, "An event-flow model of GUI-based applications for testing: Research Articles," *Software Teststing, Verification and Reliability*, vol. 17, no. 3, pp. 137–157, 2007.

[14] T. M. Cover and J. A. Thomas, *Elements of Information Theory*, 2nd ed. John Wiley & Sons, Inc., 2006.

[15] P. L. T. Pirolli and J. E. Pitkow, "Distributions of surfers' paths through the World Wide Web: Empirical characterizations," *World Wide Web*, vol. 2, no. 1-2, pp. 29–45, 1999.

[16] T. Takagi and Z. Furukawa, "Construction Method of a High-Order Markov Chain Usage Model," in *APSEC '07: Proceedings of the 14th Asia-Pacific Software Engineering Conference*. Washington, DC, USA: IEEE Computer Society, 2007, pp. 120–126.

[17] J. Pitkow and P. Pirolli, "Mining longest repeating subsequences to predict world wide web surfing," in *USITS'99: Proceedings of the 2nd conference on USENIX Symposium on Internet Technologies and Systems*. Berkeley, CA, USA: USENIX Association, 1999, pp. 13–13.

[18] M. Deshpande and G. Karypis, "Selective Markov models for predicting Web page accesses," *ACM Trans. Internet Technol.*, vol. 4, no. 2, pp. 163–184, 2004.

[19] X. Chen and X. Zhang, "Popularity-Based PPM: An Effective Web Prefetching Technique for High Accuracy and Low Storage," in *ICPP '02: Proceedings of the 2002 International Conference on Parallel Processing*. Washington, DC, USA: IEEE Computer Society, 2002, p. 296.

[20] Z. Ban, Z. Gu, and Y. Jin, "A PPM Prediction Model Based on Stochastic Gradient Descent for Web Prefetching," in *AINA '08: Proceedings of the 22nd International Conference on Advanced Information Networking and Applications*. Washington, DC, USA: IEEE Computer Society, 2008, pp. 166–173.

[21] J. Cleary and I. Witten, "Data Compression Using Adaptive Coding and Partial String Matching," *IEEE Trans. Commun.*, vol. 32, no. 4, pp. 396 – 402, 1984.

[22] R. Begleiter, R. El-Yaniv, and G. Yona, "On prediction using variable order Markov models," *J. Artif. Int. Res.*, vol. 22, pp. 385–421, December 2004.